UNIVERSITY OF OSLO
Department of Informatics

Prognostic Classification of Ovarian Cancer by LBP Texture Analysis of Microscopy Images

Master thesis

Håvard Frenning Sørensen

January 26, 2010

Abstract

Finding useful texture based features is often hard, even if there are clear patterns in the material at hand, but it will be a very useful contribution to any classification process as the texture might be uncorrelated to other more easily found features. So even if the classification rates can not compete with the ones accomplished with other approaches, they might give valuable new information.

This thesis looks at the use of the Local Binary Pattern (LBP) texture analysis technique in Prognostic Classification of Ovarian Cancer based on microscopy images of cell nuclei. The textures in this data set do not contain information that makes it possible to classify by human visual inspection. The approach used is not specific to the present data set, but could also be used on most other texture classification problems.

Differences between LBP and Gray Level Cooccurrence Matrixes (GLCM) are described, feature selection in general is discussed, and pseudocode for some of the feature selection algorithms is presented.

Extensions to regular LBP are proposed and tested together with the regular version. Grouping of the LBP data based on the nuclei sizes and gray levels and dividing into new features is tested. Other possible approaches are also outlined.

The resulting classification rates are low. The reason for this is discussed and the main theory is that too much data is included in each feature. A version of regular LBP that only looks at dark areas is tested and shows some promising results.

LBP looks like a promising tool for the future, but this thesis should give a good indication of what problems to be aware of. Any future success using LBP on this data set will probably depend on finding the areas whitin nuclei which have patterns that differ between the prognostic groups, as there does not seem to be any pattern present throughout most of the nuclei.

Acknowledgements

I first of all want to thank my supervisor Fritz Albregtsen, for all his good help and input. I also want to thank Birgitte Nielsen, Håvard Danielsen and all the other people at Radiumhospitalet who have helped me.

Contents

1 Introduction **5**
- 1.1 LBP and Texture Analysis . . . 5
- 1.2 Implementation . . . 5
- 1.3 The approach . . . 5

2 The data set **6**
- 2.1 Making of the images . . . 6
- 2.2 The image files . . . 7
- 2.3 Organization of the data set on disk . . . 7
- 2.4 Size of the cell nuclei . . . 7
- 2.5 Gray levels . . . 9
  - 2.5.1 Summing up some facts about the dataset . . . 9

3 The Methods **16**
- 3.1 Introduction to Texture analysis . . . 16
- 3.2 Gray Level Cooccurrence Matrix (GLCM) . . . 16
  - 3.2.1 Measurements . . . 17
- 3.3 Local Binary Patterns (LBP) . . . 18
- 3.4 A short comparison . . . 19
  - 3.4.1 Noise and differences in illumination . . . 19
  - 3.4.2 Rotation . . . 19
  - 3.4.3 Implementation and Efficency . . . 19

4 Feature selection **21**
- 4.1 Introduction . . . 21
- 4.2 The basics . . . 21
  - 4.2.1 What is feature selection? . . . 21
  - 4.2.2 Why do we need feature selection? . . . 22
  - 4.2.3 Combining features from different functions . . . 22
  - 4.2.4 Training and test data . . . 25
  - 4.2.5 The curse of dimensionality . . . 26
- 4.3 Preprocessing for feature selection . . . 26
  - 4.3.1 Groups of features . . . 26
- 4.4 Separability measures . . . 27
  - 4.4.1 Error probability . . . 27
  - 4.4.2 Interclass distance . . . 27

4.4.3 Probabilistic distance . . . 28
4.4.4 Probabilistic dependence . . . 29
4.4.5 Ad-hoc Criterias . . . 29
4.5 Methods for feature selection . . . 30
4.5.1 Individual feature selection . . . 30
4.5.2 Exhaustive methods . . . 30
4.5.3 Non-exhaustive methods . . . 31

5 The implementation **37**
5.1 Introduction . . . 37
5.2 The background . . . 37
5.3 Matlab . . . 38
5.4 Other possibilities . . . 38
5.4.1 Weka . . . 38

6 Results and discussion **39**
6.1 Important notes . . . 39
6.2 Parameters and approach . . . 40
6.3 Short summary of the results . . . 40
6.4 Standard LBP . . . 41
6.4.1 Motivation and weaknesses . . . 41
6.4.2 The test and parameters . . . 41
6.4.3 The results . . . 41
6.5 LBP with other radiuses and number of points . . . 42
6.5.1 Motivation and weaknesses . . . 42
6.5.2 The test and parameters . . . 42
6.5.3 The results . . . 42
6.6 Grouping based on cell nuclei size . . . 42
6.6.1 Motivation and weaknesses . . . 42
6.6.2 The test and parameters . . . 43
6.6.3 The results . . . 43
6.7 Grouping based on average gray level . . . 43
6.7.1 Motivation and weaknesses . . . 43
6.7.2 The test and parameters . . . 43
6.7.3 The results . . . 44
6.8 Using maximum value or variance . . . 44
6.8.1 Motivation and weaknesses . . . 44
6.8.2 The test and parameters . . . 44
6.8.3 The results . . . 44
6.9 Only using codes from dark regions . . . 44
6.9.1 Motivation and weaknesses . . . 44
6.9.2 The test and parameters . . . 44
6.9.3 The results . . . 45

7 Other possibilities **46**
7.1 Introduction . . . 46
7.2 General aspects . . . 46
7.3 Only comparing to good nuclei from good patients . . . 46
7.4 Only look at cells or areas with high or low variance . . . 47
7.5 Creative interpolation functions . . . 47
7.6 Grouping based on where we are in the image . . . 47

8 Conclusion **48**

A More about the application **49**
A.1 Requirements . . . 49
A.2 Easy usage . . . 49
A.3 Indata . . . 50
A.3.1 The file list . . . 50
A.3.2 The "pairs" file . . . 51
A.3.3 Supported image formats . . . 51
A.3.4 Supported mask formats . . . 51
A.4 Out data . . . 52
A.4.1 The analysis oriented format . . . 52
A.4.2 The complete format . . . 52
A.5 Special features of this implementation . . . 52
A.6 Threads and calculation speed . . . 52
A.7 Methods . . . 53
A.7.1 readAndSave . . . 53
A.7.2 readFromFolder . . . 54
A.7.3 Grouping of images . . . 54
A.8 Other functionality . . . 54
A.8.1 Feature selection and Classification . . . 54
A.9 Possible extensions . . . 54
A.9.1 Scale invariance . . . 55
A.9.2 More features . . . 55
A.9.3 Make it more automatic . . . 55
A.10 Example calculation times . . . 55
A.11 Packages . . . 56

B The Python version of the software **57**
B.1 Pyhton and Java . . . 57

C Matlab code **58**

D LBP Histograms **68**

Chapter 1

Introduction

1.1 LBP and Texture Analysis

Gray Level Cooccurrence Matrix (GLCM) and Local Binary Patterns (LBP) are two different approches to statistical texture analysis. GLCM was first defined by Haralick et al. 1973, and is in common use. LBP was first defined by Ojala et al. 1996, and is gaining popularity. The main focus in this thesis is on LBP, since GLCM already has been used for the purpose by Nielsen et al. 2004. Both methodology and results will be compared. The two functions will be more thoroughly introduced in chapter 3.

The publicly available tools for using LBP are very limited, a few hundred lines of matlab code by the University of Oulu: `http://www.ee.oulu.fi/mvg/page/lbp_matlab`.

1.2 Implementation

Program code in Java, Python and Matlab has been written as a part of this thesis in order to get a better look of what can be done with LBP. As the main focus of this thesis is not the implementation, the resulting code has some loose ends in fields that are not used in this thesis. The source code can be found at `www.freso.net/lbp` free to use. The code does not just implement the orignal LBP, but also has a lot of extensions and functionality to easier read and calculate data. More about the application can be found in chapter 5.

1.3 The approach

We will not be looking much at the medical aspects behind this thesis, there will be some few pieces of information that will be used for some deciscions and that will be a part of the argumtation. But for the most part, this thesis tries to be blind to the medicine behind the problem. The data has not been compared with data of know cancer cells by me, and no training is done with other data sets. The main reasons for this is:

- The studies this will be compared with do not look at those parts.
- It makes the task a little simpler, and makes me focus more on image analysis which is the field of the master thesis.

Chapter 2

The data set

2.1 Making of the images

134 cases of ovarian cancer classified as International Federation of Gynecology and Obstetrics (FIGO) stage I where included in the analysis [7]. 94 cases had a good prognosis, which means that they survived the follow-up period without a relapse. The minimum length of follow-up for patients alive without a relapse was ten years. The 40 cases included in the poor prognosis group died of a cancer-related disease or relapsed during the follow-up period.

Paraffin-embedded tissue samples fixed in 4% buffered formalin were sectioned ($2 \times 50\mu$m) and enzymatically digested (SIGMA protease, type XXIV, Sigma Chemical C., St. Louis, Missouri, USA) for the preparation of isolated nuclei (monolayers) [4]. The nuclei were Feulgen-Schiff stained according to an established protocol [16]. The tumor tissue to be used for the preparation were selected by a pathologist [7].

The Fairfield DNA Ploidy System (Fairfield Imaging LTD, Kent, England), which consisted of a Zeiss Axioplan microscope equipped with a 40/0.75 objective lens (Zeiss), a 546 nm green filter and a black and white high-resolution digital camera (C4742-95, Hamamatsu Photonics K. K., Hamamatsu, Japan) was used. A shade correction was performed for each image field and the image was stored in 1024 $\times$ 1024 pixels with gray level resolution of 10 bits/pixel. The pixel resolution was 166 nm/pixel on the cell specimen. Trained personnel performed a screening of the nuclei at the microscope and selected tumor nuclei for the analysis. Stromal nuclei, necrotic nuclei, doublets or cut nuclei were disregarded. The nuclei were segmented from the background by using a global threshold and stored in galleries in each case. After segmentation the cell nucleus pixels kept their gray level values i (0-1022) while the background pixel value b was set to 1023. The mean number of measured tumor nuclei/case was 281, ranging from 220 to 314 nuclei. [1]

[1]The description of the making of the images was copied from the Manuscript "Adaptive textural features from nuclear area dependent class distance matrices as putative prognostic markers in early ovarian cancer" by Birgitte Nielsen, Fritz Albregtsen and Håvard E. Danielsen

2.2 The image files

In this thesis we will use part of the L23 data set collected by Radiumhospitalet. The subset of the data set used in this thesis consists of images for 84 patients. In this subset we only have aneuploid and diploid patients.

We have a cell image and a mask image for each cell. Most of the cell images have sizes from about 1000 to 10000 pixels of which about half is the cell nucleus and the rest empty background. On average we have a litte more than one millon pixeles of data per patient.

Some samples of the nuclei images can be found in Figure 2.1 to Figure 2.4 Note that order of the images is the same in all figures.

IMPORTANT: The data in the data set is in the range 64512 to 65535, where 65535 is black and 64512 is white. This is due to the fact that the images are 10 bit grayscale images stored as 16 bit grayscale images. An histogram transform is done prior to displaying the images.

2.3 Organization of the data set on disk

In the base-folder named L23 there is one folder per patient with the patient number as folder name. Each of the patient folders contains a folder named Image. In this folder the following files per cell are used (the inconsistencies in the case of the letters is according to the data set).

L23-NNN_C_cell.tif : Contains the digital image of the cell, with pixel values is in the range 64512 to 65535.

L23-NNN_C_Mask.tif : A two-valued mask indicating if the pixel is part of the cell or the background. Pixel value is 64512 for pixels that are not part of the cell, 65535 for pixels that are part of the cell.

NNN is the three digit patient number and C is the one to three digit cell number. In the work with this thesis I have only used the data from the cell and mask images.

For most of the code I will be using folders containing data written by the Java LBP-application, the specificaton of the outformat of this application can be found in section A.4.1.

Example of a path: /L23/046/Images/L23-046_147_cell.tif (where L23 is name of the dataset 046 is the patient number and 147 is the image number).

2.4 Size of the cell nuclei

Wikipedia describes cancer as "a class of diseases in which a group of cells display uncontrolled growth (division beyond the normal limits)", therefore cell nuclei size should be of interest. The sizes of the cell nuclei is also one of the easiest measures to extract.

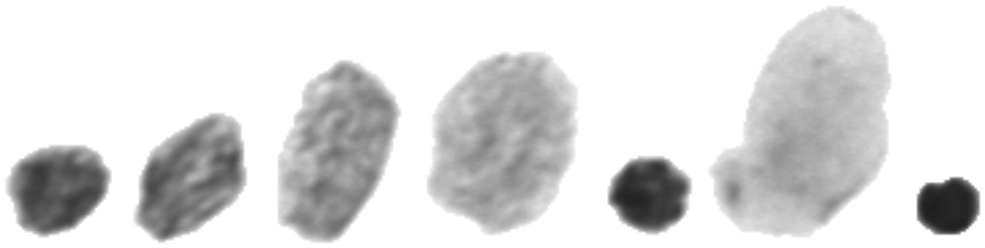

Figure 2.1: Images from one of the patients from the good prognosis group

Figure 2.2: Images from an other of the patients from the good prognosis group

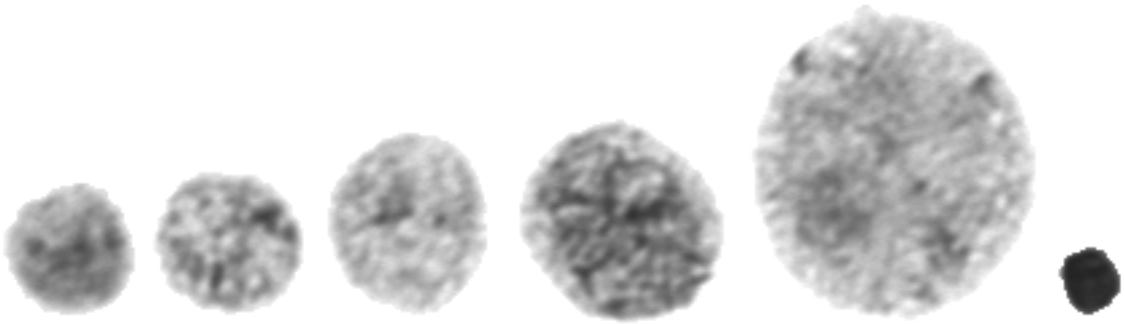

Figure 2.3: Images from one of the patients from the bad prognosis group

Figure 2.4: Images from an other of the patients from the bad prognosis group

Class	Good prognosis	Bad prognosis	Both classes
Number of patients	57	27	84
Number of images	18 091	8 783	26 874
Sum number of pixels	60 634 317	35 200 433	95 834 750
Average number of pixels	3 351.6	4 007.8	3 566.1
Maximum number of pixels	27 745	41 339	41 339
Minimum number of pixels	208	226	208

Table 2.1: Number of patients, images per patient and some statistics on the number of pixels for the good and bad prognosis class.

From Table 2.1 we see that there are differences in the average and maximum cell sizes between the groups. The differences are also in the direction we might expect, with the largest sizes beeing in the bad prognosis group.

Figure 2.5 shows how the max nuclei sizes for each patient is distributed and Figure 2.6 shows the distribution of the mean nuclei sizes. We also here clearly see the same tendencies.

Note that even though this indicates that it is possible to get some prognositic information from the cell sizes, this will not be used directly for classifing purposes in this thesis.

2.5 Gray levels

The gray levels in the nuclei is what we will be using for texture analysis, and getting a overview of their distribution might therefor be useful. Due to possible uneven coloring as a part of the process used to create these images, the variance might be a more reliable measurement than the mean gray level. Please keep in mind that higher gray levels are darker when shown in in this report (see Figure 2.1 to Figure 2.4).

Class	Good prognosis	Bad prognosis	Both classes
Mean gray level	288.0	322.7	299.4
Mean standard deviation	100.7	110.5	103.9

Table 2.2: The distribution of the grey levels within the two classes.

The average gray level and the variance in the gray level in Table 2.2 indicate that there is some differences between the classes. Figure 2.7 to 2.10 show how the gray level average and variance is distributed in the good and bad classes.

2.5.1 Summing up some facts about the dataset

There seems to be clear differences in nuclei sizes between the images in the two prognosis classes. The bad prognosis class has on average larger nuclei than the good prognosis class. The differences in gray levels does seem to be more coincidental. It should be noted that the differences in sizes between the groups indicates that other measures than those using purely texture might be more successful.

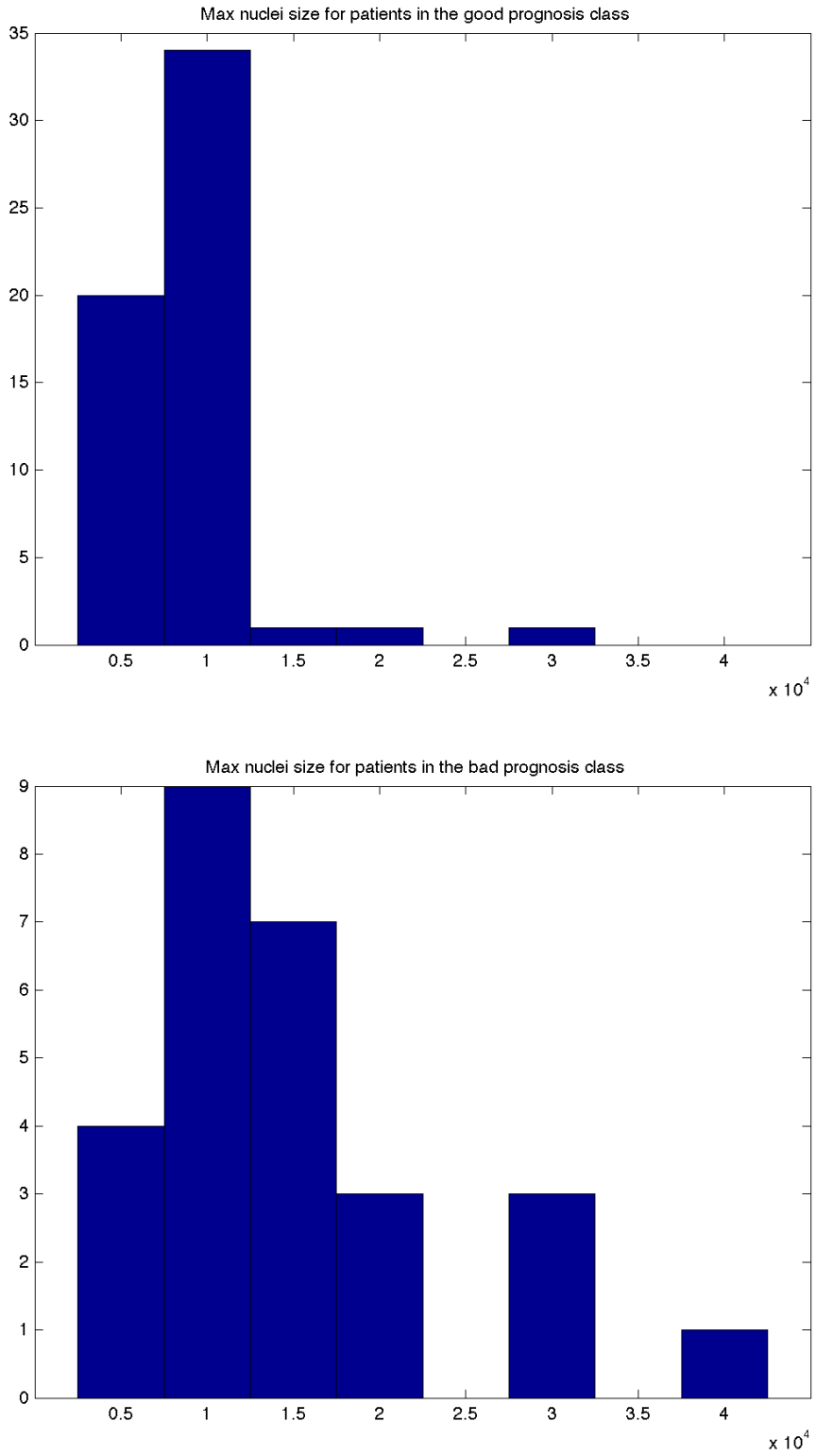

Figure 2.5: Maximum nuclei size for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

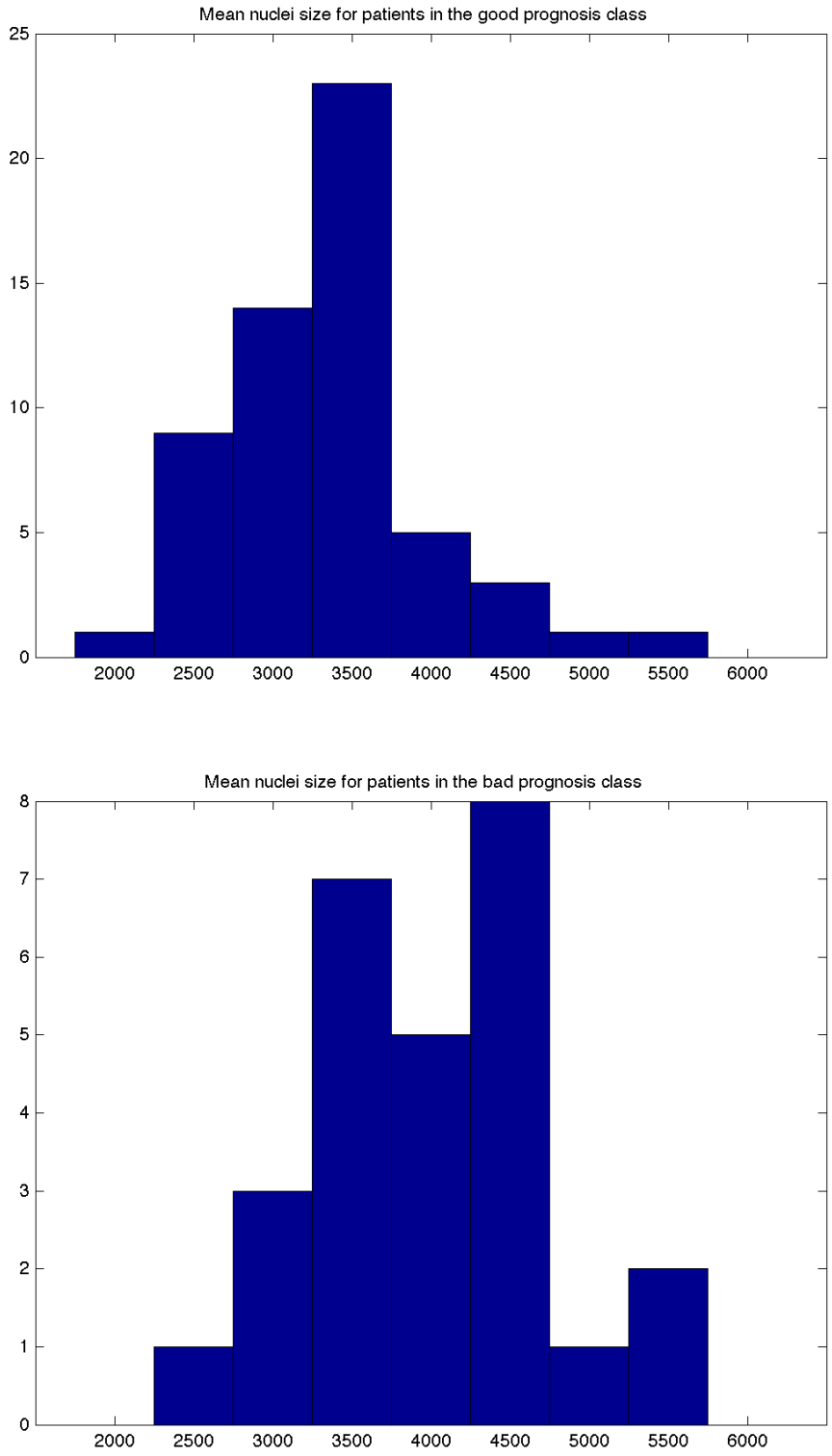

Figure 2.6: Mean nuclei size for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

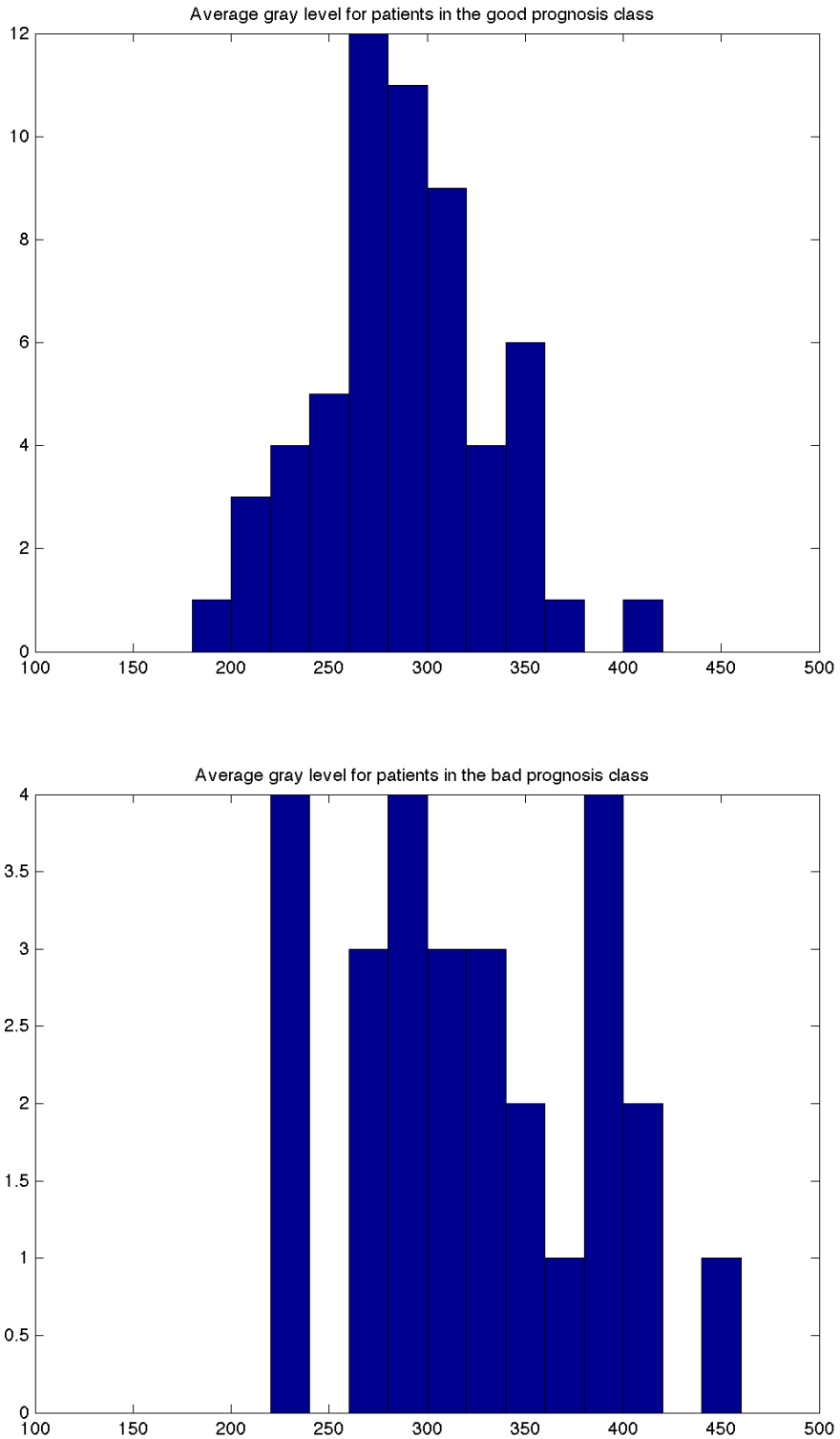

Figure 2.7: Average gray level in the nuclei for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

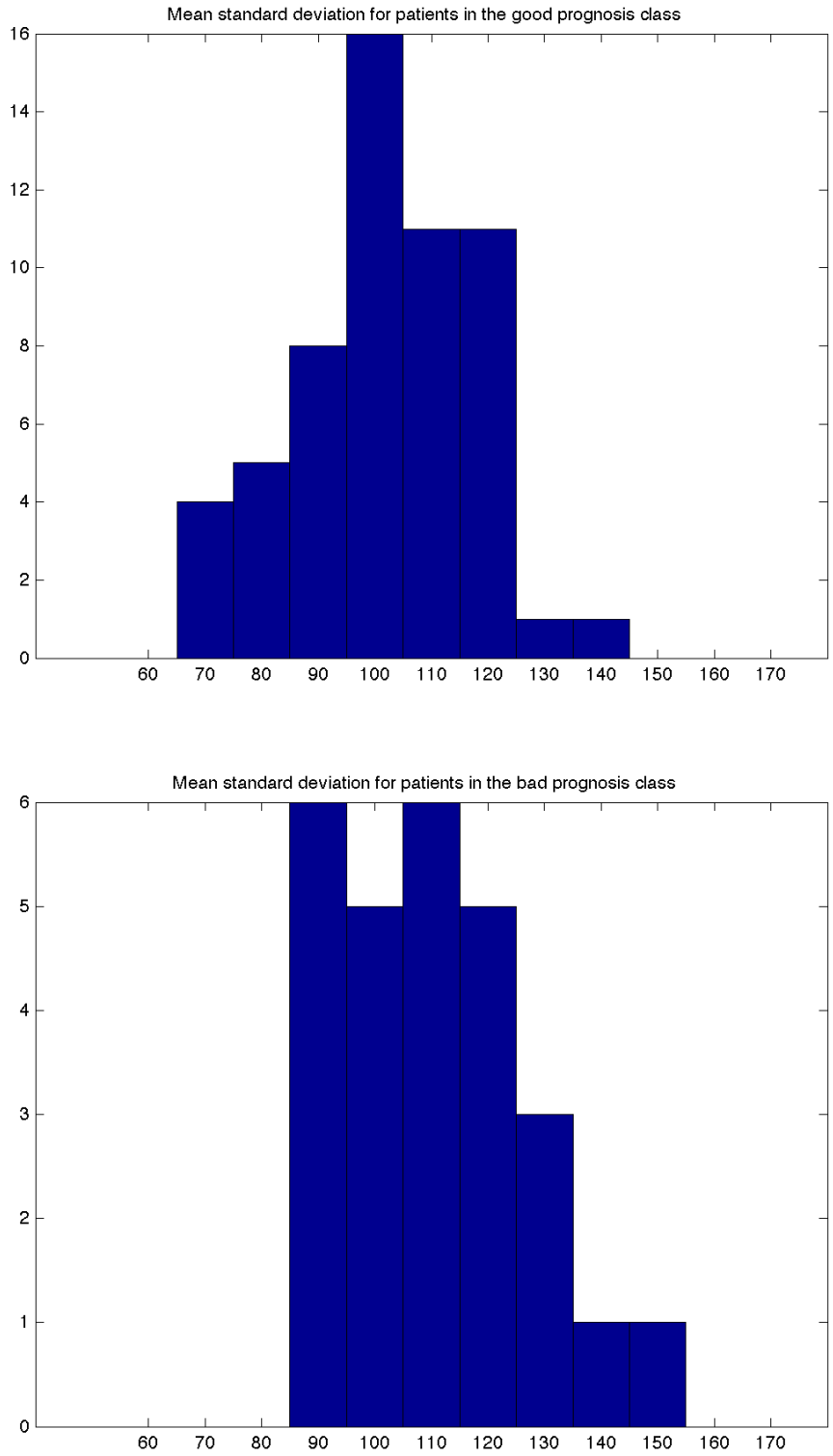

Figure 2.8: Mean standard deviation in gray level in the nuclei for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

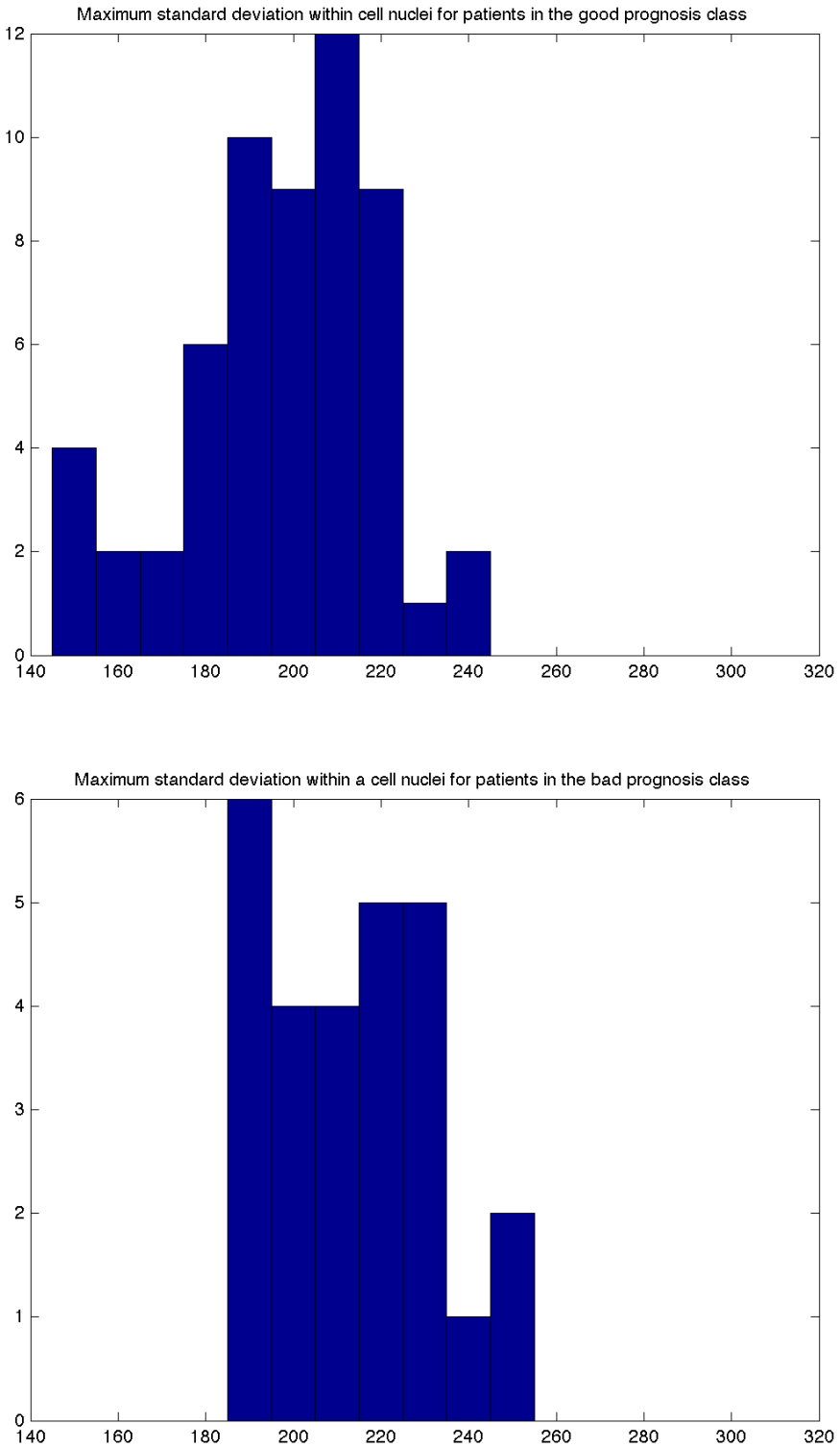

Figure 2.9: Maximum standard deviation in gray level in the nuclei for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

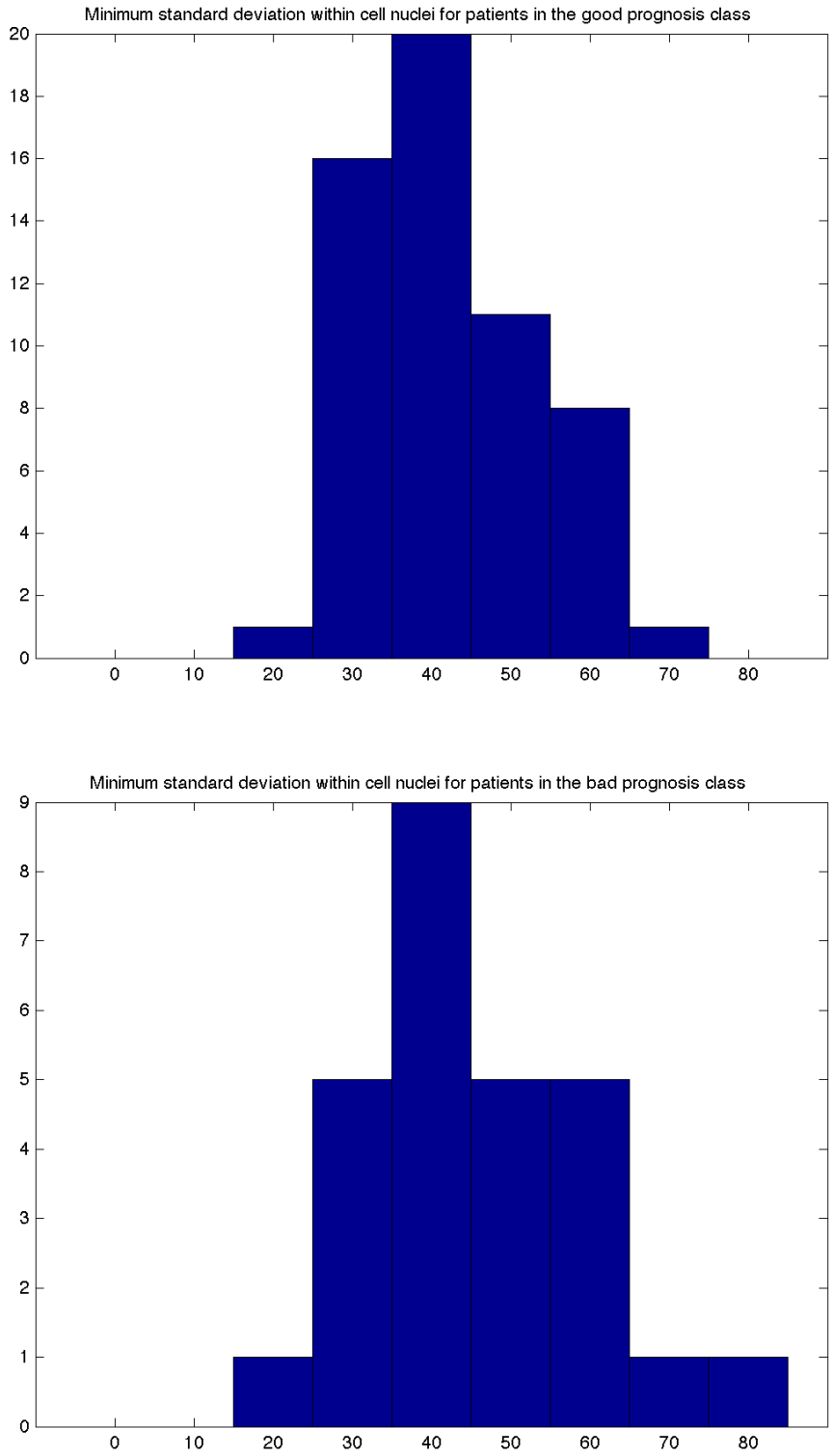

Figure 2.10: Minimum standard deviation in gray level in the nuclei for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

Chapter 3

The Methods

3.1 Introduction to Texture analysis

Textures can be defined as a function of the patterns and variations in gray levels or colors in an image. A texture is usually a pattern which to some degree repeats itself, without nessecary having anything else in common.

Both GLCM and LBP are statistical approaches, which in practice means that we collect data and then apply statistics to these collected data instead of directly calculating the values from the pixel data. In GLCM we calculate the statistics from the cooccurrence matrix, while we in LBP (most commonly) use the distribution of the different LBP-codes.

We will look more at the differences between these two approaches in this chapter.

I will first give a short introduction to both methods, and then I move on to the comparison. I will focus on the main ideas behind the methods and, where I feel that it is relevant, give some details of how they are implemented. The use on color images is not discussed.

This chapter describes parts of what I have tried and read about the different methods, not everything I mention is things I have seen in my experiments, some parts are just observations on the nature of the methods. My primary source for information on LBP is the PhD-thesis "The Local Binary Pattern Approach to Texture Analysis - Extensions and Applications" by Topi Mäenpää (Oulu University Press, 2003). The rest of the sources is mentioned under the introductions to the two methods in the next chapter.

3.2 Gray Level Cooccurrence Matrix (GLCM)

GLCM is a commonly used method for texture analysis (probably the most widespread of them all). The basic concept is that we go through (a part of) the image and for each pixel move a given number of pixels in a given direction (for example two pixels to the right) and increase the cell in a matrix which represent the transition between the two pixel gray level values. The horisontal direction in the matrix represents the source pixel values, and the vertical represents the destination pixel values.

The matrix will be of size G*G where G is the number of possible gray level values. Since G*G rapidly becomes a large number as G increases, we usually requantize the data into less than G levels. 16 is a common number of levels to requantize to, instead of 256

which is the most common number of gray levels in images. We have 1024 gray levels, so we have even more reason too requantize. Requantization gives smaller computer memory usage and removes some of the problems that noise can give. It also makes sense to reduce the number of possible matrix elements when we think about what the GLCM is going to be used for: Usually we just look at a part of the image, so if we kept the N*N size most of the matrix elements would remain zero, also in most cases there is no need to discriminate between for example gray level 117 and 118 (but sometimes there is).

The decision on how the requantization is done has a lot to say for the results. How software applications requantize the data differs, and it should also differ from one real life application to another since we are looking for different things, and the nature of the images is very different. Image quality should also be kept in mind.

For each image (or each part of the image) we can have many different GLCMs for different pixel spacing and direction. A degree of rotation invariance can be achieved by combining the matrixes for different angles (we can either combine the GLCMs them self or combine the results calculated for the GLCMs with different angles afterwards). A degree of scale invariance can be achived by making GLCMs for the same directions but different spacings.

A good basic introduction to GLCM can be found at `http://www.fp.ucalgary.ca/mhallbey/tutorial.htm`, i will list some of the measures here, together with matlab code for calculating them. A more thorough introduction to the use of GLCM in Matlab can be found in the guide "Using a Gray-Level Co-Occurrence Matrix (GLCM)" in the Matlab help system (the guide can also be found at `http://www.mathworks.com/access/helpdesk/help/toolbox/images/f11-27972.html#f11-29651`).

GLCM has already been used on the same data set as in this thesis [10].

3.2.1 Measurements

Please note that the definitions for the measures mentioned differs a little depending on where you read, there are also many different names for the same measurements. I will use the definitions found in Matlab since this i the implementation i usually use. Note that there also exist other measurements than the ones mentioned here.

Calculation of the measurements is done in Matlab by the graycoprops function, which can be called with the GLCM as first parameter and the measurement name as second parameter.

These measurements combined with a set of rules for calculating the GLCMs is what together is used as a feature.

Contrast

Contrast is a mesure of how much the gray level varies in the image. It is calculated by multiplying the elements in the GLCM with weights that is the squre of the distance between index i and j for the GLCM element. Contrast will therfore be high if we have large differences between the pixel values. Contrast i 0 for a constant image.

$$Contrast = \sum_{i,j} |i-j|^2 p(i,j) \tag{3.1}$$

Homogeneity

Homogeneity is calculated in the same way as contrast, but in stead of having weights that increase as we move away from the diagonal, we here have decreasing weights as we move away from the diagonal. In matlab homogeneity will be a value in the range 0 to 1 (both limits included).

$$Homogeneity = \sum_{i,j} \frac{p(i,j)}{1+|i-j|} \tag{3.2}$$

Correlation

Correlation measures the amount of dependance between pixels and the neighbouring pixels. Correlation is in the range from 1 for a perfectly positively correlated image and down to -1 for a perfectly negatively correlated image. The Matlab implementation will return NaN (Not a Number) for a constant image.

$$Correlation = \sum_{i,j} \frac{(i-\mu i)(j-\mu j)\vec{p}(i,j)}{\sigma_i \sigma_j} \tag{3.3}$$

Energy

Energy is the sum of the squared elements in the GLCM, this will give high values if the same gray level transitions occur often. In matlab the value is mapped to the range 0 to 1 (both limits included).

$$Energy = \sum_{i,j} p(i,j)^2 \tag{3.4}$$

3.3 Local Binary Patterns (LBP)

The basic idea is that we choose a pixel which we call the center pixel and then pick P pixels on a circle with radius R around the centre pixel. Each of the P pixels is compared to the center pixel and the result is a binary number (for each pixel we have 1 if the gray level is higher than or equal to the center gray value, and else 0). We move counter-clockwise, starting with the rightmost pixel for the least significant bit.

When using large radii the results can get pretty random. Therefore it can be a good idea to use some kind of filtering (not only to handle noise, but also so that the pixel value is more like a weighted average of the sorrounding pixels) to make the results more useable. For example a Guassian low pass filter can be used.

For each center pixel we can have many different LBP codes (with different R and P values) and using different versions of the image can also be an option (for example versions where some kind filter has been applied).

Local Binary Patterns (LBP) has shown good results in many applications and is gaining popularity. LBP is often much faster than other methods and handles noise and differences in illumination well in most cases [1].

[1] all this according to "The Local Binary Pattern Approach to Texture Analysis - Extensions and Applications" by Topi Mäenpää (Oulu University Press, 2003)

Features are usually extracted from the LBP codes by making frequency tabels for the number of times one of the LBP code is observed for each image or region. Unless all images (or regions) are of the same size we need to normalize the data (so that the sum of all frequencies for the image or region is 1) before storing it in the frequency table.

3.4 A short comparison

The two most obvious differences between the two methods are:

- LBP looks at one pixel and it's relation to several other pixels, while GLCM always compares two and two pixels (this is at least the case with the basic implementations, and the basic idea).
- LBP does not care how big the difference in gray value is, while the basic idea behind GLCM is to compare the magnitude of the differences.

3.4.1 Noise and differences in illumination

The fact that LBP compares many pixel values against one pixel value (the center pixel), means that if the value of the center pixel is incorrect or atypical (due to noise, or other type of atypical value) this will have a dramatic impact on the result. Some kind of filtering (or requantization) may therefore be needed in order to get useful results.

LBP completely ignores the magnitude of the gray level difference since it was designed to be an addition to methods focussing on this. The variance in the neighborhood, can in many cases be a good supplement when information about the contrast is needed.

GLCM on the other hand is very dependent on the magnitude of gray level differences. If not used with care, noise and changes in illumination can make the results worthless. These issues can be handled with filtering of the source image, or post processing of the GLCM, but in many cases you find that this removes a lot of the magnitude information, and you will get the same results with LBP, but with easier implementation and lower computation time.

3.4.2 Rotation

LBP codes can be made rotation invariant, by doing circular bit shifting on the LBP code until it reaches its lowes possible value (it then becomes $(360°/P)$ rotation invariant). This bit shifting can be done very efficently in many programming languages, but the best choice for larger calculations is to use a lookup table as each LBP code only has one corresponding rotation invariant code.

GLCMs is usually made 180° rotation invariant by counting element[x,y] and element[y,x] in the matrix together. Higher degree of invariance can be achieved by combining the matrixes for different angles.

3.4.3 Implementation and Efficency

Both methods can of course be implemented in a number of ways, and I see no reason to compare the speeds of the MATLAB functions against each other (they are developed by

different people). I will however try to say something about the complexity of the two on an image with width N and height M. The fact that we can not calculate the codes for pixels near the borders is left out of the calculations.

Calculating the LBP codes using P points on the circle takes P*N*M comparisons. If any kind of interpolation is used this will give a significant increase in computation time. Memory usage for storing the result is N*M (or if we store the frequence table we will need one space per possible LBP code).

Making the GLCM matrix needs A*N*M subtractions (with 3 array access operations each). Memory usage for storing the result for G gray levels is G*G*A.

Chapter 4

Feature selection

4.1 Introduction

Usually we need to select from features based on different methods, but in this case we will only be selecting from LBP-based features. We need to:

1. Find the best parameters to LBP
2. Use the results from the LBP function to make features (most of the time we will only use the frequencies for the different LBP-codes, but other approaches might also be used)
3. Find the features that give the best combined results

It is part 3, finding the features that give the best combined results, which usually is referred to as feature selection. If LBP gives good results, the features found will then afterwards be combined with features using other methods.

The human visual system is most of the time not capable of discriminating between the classes (some of the cases are easy to classify, while most seem impossible to discriminate for humans). Therefore we have to expect high error rates. Based on results from other studies on the same material, a classification rate of 70% is very good, even though we only have two possible classes (good or bad prognosis).

The number of LBP combinations are 256 for regular LBP and 36 for the rotation invariant standard LBP. Even though these are not large numbers the number of possible combinations are still much too high to do exhaustive search. The number of possible combinations for 36 features is $36^2 = 68719476736$. But if the number of features can be reduced using other measures, exhasutive search on the remaining candidates is manageble, if 10 features remain we only have $10^2 = 1024$ combinations to test when doing exhaustive search.

4.2 The basics

4.2.1 What is feature selection?

Feature selection is the process of choosing a subset of all available features for use in a decision process. Feature selection is a part of most projects in automated image analysis.

Each feature needs to be tested, either by itself, as a part of a group of features or both (testing groups of groups might also be useful if we have some information about features that should or should not be used together, see section 4.3.1).

4.2.2 Why do we need feature selection?

Feature selection is needed since we usually have a lot more possible features than are practical in most applications. Having too many features does not only imply more work and higher computation times, it might also degrade the classification accuracy. This is due to overfitting to the training set or because we fail to give the really good features enough weight (the curse of dimensionality, see section 4.2.5) .

A common misconception is that the more features we use in the classifier, the better. This might be based on the following faulty theory: "If a feature is random, it will do no harm, since the random features will even themselves out, and all other features will be a positive contribution". The flaw in this theory is that it is missing the fact that the variance will increase, the following example illustrates this:

An example

We have the following features:

- Ten normally distributed features, all with expectation value 1 and variance 100.
- One normally distributed feature with expectation value 10 and variance 10.

Positive values indicates that we will classify the sample correctly. The sum of the ten first featuers and the value of the second single feature will both be 10 in the average case.

A Matlab-code doing some calculating on this example with 10 000 values of each feature can be found in program code listing 1. The code shows that the averages are almost equal as expected (see comments in the code), however the classification rates are very different.

The results can be found in Figure 4.1.

4.2.3 Combining features from different functions

In most cases we do not only get features from one method, but from several, and that is also the case when it comes to the possible real life application of this master thesis. The results from the use of LBP will be combined with the results of other methods by using some features from each, combined into a complex classifier. So the feature selection will be done in two different stages:

1. Find the best LBP-features (in order to see if LBP might be a candidate for real life application, and since I do not have access to all the other features)
2. Find the best of all the features (in order to use the best ones in the real life classifier)

Program 1 Matlab code illustrating the problem that occurs if we add too many features

```
% expectation value and variance for the 10 features
ex10 = 1;
var10 = 100;

% expectation value and variance for the last
ex1 = 10;
var1 = 10;

% number of tests
n = 10000;

sum10 = sum(normrnd(ex10, var10, 10, n));
sum1 = normrnd(ex1, var1, 1, n);

% histograms, see figure
hist(sum1)
hist(sum10)
hist(sum1 + sum10)

% printing the averages
sum(sum1 + sum10)/10000 % returns 20.1868
sum(sum1)/10000 % returns 10.0287
sum(sum10)/10000 % returns 10.1580

%printing the classificaton rates
sum((sum10 + sum1) > 0) / 10000 %returns 0.5258
sum(sum1 > 0) / 10000 % returns 0.8371
sum(sum10 > 0) / 10000 % returns 0.5123
```

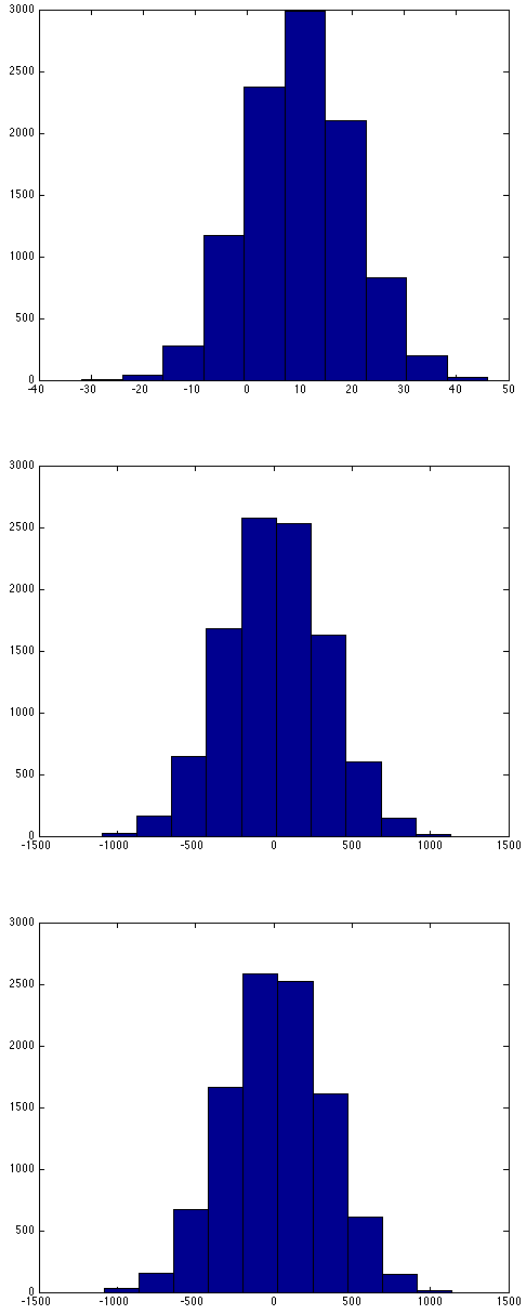

Figure 4.1: Histograms for the distributions, using 10000 numbers per feature. Only the last feature is basis for the histogram on the top, the ten others on the middle one, and the combination of all on the bottom. Even though the averages for the two top histograms are nearly identical (10.0287 and 10.1580), the correct classification rates are very different (83.7% and 51.2%). The bottom histogram has an average value of 20.1868, which is twice as much as the two others, but since we still have a too high variance the correct classification rate is only 52.6%.

One should note that the best selection of LBP-features, might not be the ones that work best together with other features. If for example both an LBP-feature and another feature have the same weaknesses, we will be more sensitive to this error. We might leave out other features that would have given a better contribution and we will also in many cases overestimate the probability of correct classification.

So even though we will end up with a list of the "best LBP-features", this does not necessarily mean that these are the ones that should be used in the end product where also other classes of features will be used.

Also note that all of the algorithms described is meant to be run on a set of all the candidate features.

4.2.4 Training and test data

Many features will need to be trained on a training set before they can be used (this is the case with both LBP and GLCM features). After the features have been trained we use a separate test set in order to evaluate their performance. We can divide the data into two fixed sets, and use one for training and the other for testing. If we have large amounts of images this is a usable approach, but in most cases we do not have enough test data to get reliable results with fixed sets.

In order to get more reliable results with limited amounts of available test data we can run multiple rounds of training and testing where the test data is divided in several ways. When we do this we can also calculate the variance in the classification results. When we divide the images in test and training set we are free to choose the sizes of both sets. One popular approach is "Leave one out" in which we train with all images, except for one which we try to classify. This process can be repeated until we have used all images as the test image once.

The test set should never contain any of the pictures that was used to train the features. Not using separate test and training set, will give an unrealistic test and will in most cases give an overestimate of the classifiers strength. Randen and Husøy also found that failing to use different test and training sets might cause the wrong features to be selected since the overestimation varies from feature to feature [13].

Another important point is that we never should alter the test or training sets based on the performance of the features, there will often be images we are tempted to remove in order to gain better results. If images are removed this has to be documented (and preferably a method for detection of this type of data proposed).

The general rule is: The more features we want to use, the more test data we need to use. So if the amount of test data is limited we also need to limit the number of features.

IMPORTANT: In our data set there is 57 patients in the good prognosis class and 27 in the bad prognosis class. Usually we dont need to have the same number of each feature, but since we here typically get classification rates that are worse than the ratio between the classes, feature selection ends up including features that puts 99% in the dominant class. This is not desirable and, we therefore need to keep the number of patients from each class about even.

4.2.5 The curse of dimensionality

In addition to the more obvious problems associated with the huge search space, like not having the ability to perform exhaustive search and long computation times, we can also have problems with coincidental "good" feature combinations. These are combinations of features that match the test set very well, but that do not generalize to new data.

The term "Curse of Dimensionality" was coined by Richard Bellmann and describes the problem that occurs when we get exponential increase in volume (here possible combinations of features) when we add extra dimensions to a mathematical space (here adding features). We might test thousands (or millions) of feature combinations, finding combinations that work very well on our test set. But it is a real possibility that the selected combination will fail on a test set.

In addition to reducing the number of features tested, we can also increase the number of test images to remedy this, but in most cases we are already using all available test images.

Note that this is not the same as having overfitted features. When we have features that are overfitting to the data set this is due to features that adapt to much when they are trained. This can be avoided by using separate training and test sets.

4.3 Preprocessing for feature selection

We usually need to limit the number of features, for reasons already mentioned in 4.2.2 or others, but then we need each feature to be as good as possible. Noise in the data sets may degrade the accuracy. We need to remove the noise. This can either be done as a part of the feature, we can make features that only look at some of the data, or we can modify the data set before we calculate the features.

What is considered as noise in some applications might be useful for calculation of features in others, and what is noise might also differ from feature to feature in the same applications. There are no definitive answers, other than that you need to be aware that there will be some noise in most data sets and you might need to do something about it.

The methods for handling these difficulties are the same as in statistics: outlier removal and data normalization. In most cases the amount, type and handling of noise in the data set should reflect how it will be in the real life application.

Also removing obviously weak features and grouping of similar features to identify the ones with the same weaknesses might be good strategies.

4.3.1 Groups of features

We start with a lot of different feature measurements. These then need to be used together in a useful way in order to get a good classifier.

The different types of features might have different strong and weak points. When it comes to LBP, handling lightning differences can be mentioned as a strong point and color support might be a weak point. The strong and weak points might also vary depending of implementation and based on differences in parameters (many implementations of LBP do not consider color information at all, others do).

It might be a good idea to separate the different features into groups by their known weak and strong points, so that we can make sure we do not select too many features with the same weaknesses (we need to be aware of the amount of test and training data that triggers this weakness). Other kind of information that we get from how the features are calculated can also be useful to include in the process of selecting features.

Dividing the features into groups is usually not a part of the feature selection algorithm, but something we do before we decide which features to test. This step usually comes naturally, since we often feel that adding more features of the same kind can not help much. How this should be done, or if it should not be done at all depends on which feature selection algorithm that will be used, the criterias that will be used, the number of features and the quality and amount of test data. The feature selection algorithms are usually good at picking only the best of similar features, but might not have all the information you have or might need too much time to test all possibilities.

4.4 Separability measures

We need to have some measurements of how good the features are, both individually and together. Most of the following approaches can be used both as measure of a single feature and of a collection of features.

This is far from a complete list, for more complete lists see [3], [18] or other sources. Also other kinds of measures might be useful depending on the problem at hand.

4.4.1 Error probability

Using the error rate is the most intuitive measure since we want features that lead to good decisions (which is decisions with low error rates). It can be estimated by using test data, but the amount of test data will often limit the accuracy. The computation times will also be higher than for most other measures. Error probability can be expressed as

$$\int [1 - \max_i P(\omega_i|\varepsilon)] p(\varepsilon) d\varepsilon$$

where ε is the feature vector with the candidate features and $P(\omega|\varepsilon)$ is the a posteriori probability function.

4.4.2 Interclass distance

This measure looks at the pairwise distance between the elements of different classes in a space (often 2 dimensional). In doing so it does not rely on an underlying probability density function. The distance can be be measured by any distance function we want. Euclidean distance will probably often be the first choice, but sometimes we might want to give more weight to differences in some directions in the space more than others. Sometimes we might also want nonlinear functions to give more or less weight to the extremes.

4.4.3 Probabilistic distance

Probabilistic distance is a measure of the amount of overlap between two probability density functions. Any function having its max value when the functions are disjoint, minimum value when they are equal and never has negative values can be used as an probabilistic distance measure. A small selection of distance measures will be described in this chapter. Other measures can also be used, for a more complete list see other sources ([18] has a good collection).

Both the parametric form and the original definitions are listed for some measures even though we for the most part only use the parametric form. The reason for doing this is that the original definitions are easier to understand, while the parametric forms are easier to compute. Averaged versions of the distance measures are not listed. We can get the averaged versions by taking the a priori probabilities into account, this is done by multiplying the conditional probabilities $p(\varepsilon|\mu_i)$ with the a priori probabilities $P(\mu_i)$.

In the following subsections ε is the feature vector with the candidate features, and ω_i denotes class i. For the parametric forms μ_i is the mean vector and Σ_i is the Covariance matrix for class i.

Euclidean distance

Euclidean distance measures what we usually intuitively consider as distance. The distance is not scaled, and the variance in the different classes is not considered.

$$(\mu_2 - \mu_1)^T(\mu_2 - \mu_1)$$

Mahalanobis distance

Mahalanobis distance differs from Euclidean distance in that it is scale invariant. The distances are scaled by the variance in the data set.

$$(\mu_2 - \mu_1)^T\Sigma^{-1}(\mu_2 - \mu_1) \text{ if } \Sigma_1 = \Sigma_2 = \Sigma$$

If Σ_1 and Σ_2 is not equal Bhattacharyya distance (next section) is a good choice. If Σ is equal to the identity matrix, Mahalanobis distance will be the same as Euclidean distance.

Bhattacharyya distance

Bhattacharyya distance looks at both the scale invariant distance and differences in variance. The effect of this is that two classes with the same center will still have a distance if the variance is different.

$$-ln \int [p(\varepsilon|\omega_1)p(\varepsilon|\omega_2)]^{\frac{1}{2}} d\varepsilon$$

which has the following parametric form:

$\frac{1}{4}(\mu_2 - \mu_1)^T[\Sigma_1 + \Sigma_2]^{-1}(\mu_2 - \mu_1) + \frac{1}{2}ln[\frac{|\frac{1}{2}(\Sigma_1+\Sigma_2)|}{\sqrt{|\Sigma_1|\Sigma_2|}}]$

Divergence

Divergence is another measure of the difference between two different probability distributions. Divergence is defined as:

$\int[p(\varepsilon|\omega_1) - p(\varepsilon|\omega_2)]ln[\frac{p(\varepsilon|\omega_1)}{p(\varepsilon|\omega_2)}]d\varepsilon$

which has this parametric form:

$\frac{1}{2}(\mu_2 - \mu_1)^T(\Sigma_1^{-1} + \Sigma_2^{-1})(\mu_2 - \mu_1) + \frac{1}{2}tr\{\Sigma_1^{-1}\Sigma_2 + \Sigma_2^{-1}\Sigma_1 - 2I\}$

tr is the trace, the sum of the values on the main diagonal: $tr(A) = a_{11} + a_{22} + ... + a_{nn} = \sum_{i=1}^{n}(a_{ii})$.

4.4.4 Probabilistic dependence

Probabilistic dependence looks at the difference between the conditional and the general probability density functions. The larger the distance, the better the chance of good classification. The same distance measures as for probabilistic distance (section 4.4.3) can be used. Instead of two different conditional probability density functions we use one conditional and one general probability density function. Using divergence (see 4.4.3) we get

$\sum_{i=1}^{n} P(\omega_i) \int[p(\varepsilon|\omega_i) - p(\varepsilon)]ln[\frac{p(\varepsilon|\omega_i)}{p(\varepsilon)}]d\varepsilon$

where n is the number of classes. This is also is known as Joshi dependence.

4.4.5 Ad-hoc Criterias

Other criterias might be of importance. Many different factors can be of importance in some projects, examples include:

- Computational speed (some features are easy to compute, others almost impossible)
- Size of program code (probably not so often relevant any more other than as a measure of how easy it is to maintain)
- Cost of program code (we may be using costly software to compute some features)
- Special hardware needs (if the feature needs an extra censor this can be expensive)

If we have requirements like these, adding these to the separability measurements as weights or an integrated part of the measure can be an good choice.

4.5 Methods for feature selection

The easiest way of selecting features, called individual feature selection, is looking at them one by one. We calculate some measure of how good each feature is and then afterwards pick the ones that give the highest scores. But this is usually not the best combination of features [2]. The main problem with this approach is that many of the features are correlated and adding a new one might not even give any new information at all, instead we might get some of the problems mentioned in section 4.2.2.

To avoid these effects we need to look at combinations of the features, the feature vectors. This is called multiple feature selection. Doing an exhaustive search might not be possible depending on the size of the search space, so both exhaustive and non-exhaustive methods will be described in the following sections.

We can use individual feature selection first to find the most promising features, and then use one of the methods for multiple feature selection to find the best combination of these.

4.5.1 Individual feature selection

Individual feature selection is fairly straight forward:

1. Decide on a way to measure the performance of the features
2. Calculate the value of this measure for all features
3. Remove the features with the lowest scores

Many different measures can be used, both those mentioned in section 4.4 and others ([11] mentions some).

4.5.2 Exhaustive methods

Finding the optimal solution is in most cases not an option because of the extreme rate the decision tree grows at, but in some cases when we have few features it might be possible. When possible in an acceptable amount of time we should always use one of the exhaustive methods.

In order to find the optimal solution we have to do an exhaustive search through all the possible combination. This can be done with many different search algorithms, depth-first search, breadth-first search and others. Using methods like branch and bound is often a good choice since we then can reduce the search space by omitting clearly non-optimal combinations.

Since the exhaustive methods always finds the optimal solution, the details of their implementation is not of significance for other reasons than calculation speed. I will therefore not go in to the details of their implementation, but these can be found in books focusing on general computer algorithm design. Descriptions of all the mentioned exhaustive methods can be found in [1].

4.5.3 Non-exhaustive methods

In most cases we have to use one of the non-exhaustive methods, because of the huge number of possible combinations.

All of these methods need to use some kind of criteria for how good the solution is, this can be one from section 4.4 or any other criteria you find appropriate.

On the following pages different search approaches are described. The methods is roughly ordered by the complexity of their algorithms. Note that even though some of the methods are better than others in the average case, this does not mean that it is the best in all cases. The original method might give better results than an improved version. This is due to the fact that none of the following methods goes through all the possible combinations, and different combinations are left out by the different methods.

Sequential backward selection

In sequential backward selection [8] we start with all the features and remove one by one until we have our subset. Program listing 2 illustrates the algorithm.

Program 2 Psudo code for sequential forward selection

```
# all arrays/lists are indexed from 0

features = <array/list containing all features>
remove = -1;
betterSolution = true;
bestSolutionValue = 0;

while (betterSolution) {
    for (<all i between 0  and length of features - 1>){
        value = <calculate the result for features excluding feature
                at index i in feature using some criteria>
        if (value > bestSolutionValue){
            bestSolutionValue = value;
            remove = i;
        }
    }

    if (remove != -1) {
        <remove the feature at index remove>
        remove = -1;
    } else {
        betterSolution = false;
    }
}
```

The code above continues as long as one of the reduced sets is a better solution. Another common way of deciding when to stop is to set a fixed number of wanted features

and stop when the set is reduced to this size. This approach is used in the next code example for sequential forward selection.

Sequential forward selection

In sequential forward selection [17] we do the the reverse of the sequential backward selection. The approach is the same, but instead of removing features we add them one by one.

Program 3 Psudo code for sequential backward selection

```
# all arrays/lists are indexed from 0

allFeatures = <array/list containing all features>
features = <empty array/list>
add = -1;
maxNumberFeatures = <maximum number of features>

while (<length of features> < maxNumberFeatures) {
    bestSolutionValue = 0;
    for (<all i between 0  and length of allFeatures - 1>){
        value = <calculate the result for features including feature
                at index i in allFeature using some criteria>
        if (value > bestSolutionValue){
            bestSolutionValue = value;
            add = i;
        }
    }

    <add the feature at index add in allFeatures to feature>
}
```

This code in listing 3 will add one and one feature until maxFeaturesWanted is reached (note that I in the code for sequential backward selection have added a small modification to the basic idea which also can be used for this method).

Sequential forward selection is faster than sequential backward selection when we want to keep more than half the features, otherwise sequential backward selection is the fastest. Which one that gives the best solution will differ, but none is better than the other in the average case.

Add and remove

Both sequential forward selection and sequential backward selection both have what we call the nesting problem [3]. If a feature is added or removed this decision will not reconsidered. Since this in many cases can be a bad policy we also have search strategies

without these restrictions, they both add and remove. The add and remove approach has has been refined and subtyped into a lot of methods.

In the average case the add and remove methods are better than the sequential ones, but at the cost of a higher computational complexity.

The add and remove methods can be implemented as primary forward or primary backward. In most cases using the primary forward approach is best if we need less than half the features, while the primary backward is best in the other cases.

Floating search

Floating search [12] is one of the more popular algorithms for an add and remove approach. Jain and Zongker [6, 19] have found this to be the best of the of the non-exhaustive methods for finding the best solution. Pseudo code for floating search (implemented as primary forward) can be found in program code listing 4

Like most other methods floating search also have some variations. Adaptive floating search is a subtype that might be of interest [15], in many cases it finds a better solution, but at the cost of added algorithm complexity.

Oscillating search

While the other methods already described either start with an empty or complete feature set, oscillating search start with a set of the desired size d. Then we repeatedly alternate between adding and removing features (while the number of feature oscillate). Oscillating search uses a δ specifying the maximum distance for the number of features from the desired d-value (this δ restricts the algorithm from evaluating too small or too large subsets during the whole search process, not only the result). More details and description of the algorithm can be found in the article [14] where the algorithm was first introduced.

Other search methods

There are also versions of sequential forward selection and sequential backward selection that add more than one feature at a time, these versions are called the generalized versions. This idea can be adopted also for add and remove methods, and might be useful when making an ad-hoc method.

Some times you have information that does not exist in the general cases and that the general algorithms do not consider, but might be of interest for a algorithm searching for the optimal solution (or an sub optimal one).

This information might make it easy to reduce the number of features down to a small enough number that the optimal search approaches are possible to compute in a reasonable time. It might also be easily incorporated into one of the suboptimal search approaches making them more likely to find a better solution (in many cases this will just be using ad-hoc criteria, see 4.4.5). In some cases the extra information requires you to make your own method in order to get full use of it.

Program 4 Psudo code for floating search

```
# all arrays/lists are indexed from 0

maxNumberFeatures = <maximum number of features>
features = <The result of running sequential forward selection
            for 2 features>
otherFeatures = <list/array containing all features not in
                 features>
temp = <two dimensional list/array>
temp2 = <two dimensional list/array>
tempValues = <one dimensional list holding the values for the
              values got when evaluating the subset of
              features on the corresponding in temp against
              a criteria>
k = lenght of features; //number of features in the current subset

while (lenght of features < maxNumberFeatures) {
    // Step 1

    bestSolutionValue = 0;
    add = -1;
    for (<all i between 0 and length of otherFeatures - 1>){
        value = <calculate the result for subset at index k in temp
                 including feature at index i in otherFeatures
                 using some criteria>
        if (value > bestSolutionValue){
            bestSolutionValue = value;
            add = i;
        }
    }

    temp[k + 1] = <new list/array equal to temp[k] but also
                   including the feature at index add
                   in otherFeatures>


    // Step 2

    bestSolutionValue = <some kind of max value>
    remove = -1;


# continues on next page
```

Program 5 Psudo code for floating search continued

```
# continued from previous page


    for (<all i between 0 and length of temp[k + 1] - 1>){
        value = <calculate the result for subset at index
                k + 1 in temp excluding feature at
                index i, using some criteria>
        if (value > bestSolutionValue){
            bestSolutionValue = value;
            remove = i;
        }
    }

    if (remove == k + 1) {
        k = k + 1;
        break; // start at step 1 again
    } else if (bestSolutionValue < tempValues[k]){
        break; //go to step 1 again
    } else if (k = 2) {
        temp[k] = <list equal a copy of temp[k + 1] but excluding
        feature remove>
        tempValue[k] = bestSolutionValue;
        break; // back to step 1
    }


    // Step 3
    while () {
        temp2[k] = <list equal a copy of temp[k + 1] but excluding
                    feature remove>

        bestSolutionValue2 = <som kind of max value>
        remove2 = -1;


# continues on next page
```

Program 6 Psudo code for floating search continued

```
# continued from previous page


        for (<all i between 0 and length of tempK - 1>){
            value = <calculate the result for tempK excluding
                    feature at index i, using some criteria>
            if (value > bestSolutionValue2){
                bestSolutionValue2 = value;
                remove2 = i;
            }
        }

        if (bestSolutionValue2 < tempValues[k - 1]){
            temp[k] = tempK;
            break; // back to step 1
        }

        temp2[k - 1] = <a copy of temp2[k] with the item on
                       place remove2 removed>
        k = k - 1;

        if (k = 2) {
            temp[k] = tempK;
            tempValue[k] = bestSolutionValue2;
            break; // back to step 1
        }
    }
}
```

Chapter 5

The implementation

5.1 Introduction

In order to test LBP we need an implementation of the algorithm, as mentioned in chapter 3, a Matlab implementation already existed, but the functionality was very limited. I was not able to find any other openly aviable implementations, I therefor decided to implement

The application it self is descibed in appendix A. The appliplication can be found at: `www.freso.net/lbp`

5.2 The background

Since my experience with matlab was limited I did not want to develop the application in Matlab. I decided to try developing it in Pyhton instead, even though my experince with Python also was limited, but Python seemed to be better for projects of this size and also faster than Matlab. Python is also a free language, which is not the case with Matlab.

The Python implementation is fully functional for the basic calculations (and contains more functionalty than the Matlab functions), it is however not as fast as hoped, and because of my limited Python knowledge I felt that it would be better if I switched language. The Python version of the software is however accessible on the web free for any use for anyone:

`www.freso.net/lbp/python`

My programming language of choice is Java, this allowed me to write faster code (Java is also sigifically faster than Python in the general case) and code more in the "spirit" of the programming language. It is the Java version that is discribed in this chapter. Some more information about the Python version can be found in appendix B

The Java version is also freely aviable on the web, but some parts might have been removed since they are specially made for use with the data formats at RadiumHospitalet:

`www.freso.net/lbp/java`

More about the choice of language can be found in appendix B.1.

5.3 Matlab

All feature selection and classification is done in Matlab. Matlab has some functionality in its Statistics toolbox that can be used for this.

5.4 Other possibilities

5.4.1 Weka

Weka could have been used in stead of Matlab for feature selection. Weka is a free collection of machine learing algorithms in Java, see their web site for more details:

`http://www.cs.waikato.ac.nz/ml/weka/`

It did however seem to be a little harder to use than Matlab for the tasks I wanted. Since Matlab also is more widely used in the field, I chose this for the task.

Chapter 6

Results and discussion

6.1 Important notes

Please note that most calculations are made with code written as a part of the work with this thesis, and any conclusions therefor build on the assumption that this code is correct. I have done my best to check for errors, so there are hopefully no errors that significally alters the results.

When listing the results a positive outcome is belonging to the good prognosis class, and negative is belonging to the bad prognosis class. Specificity and specificity is defined as follows:

$$Specificity = \frac{TrueNegative}{TrueNegative + FalsePositive} \tag{6.1}$$

$$Sensitivity = \frac{TruePositive}{TruePositive + FalseNegative} \tag{6.2}$$

Many of the tests group the images in groups based on size (and in other average gray level). In order to keep the program code simple, I have chosen to remove the groups that do not contain data for all patients. This might lower the classification rate, but it also removes the need to do suboptimal choises at other points (also the built in functions in Matlab does in many cases not support matrixes with missing data).

To make things as easy as possible a Matlab script is used to run the tests, see code listing 13 in appendix C. All results should be possible to improve by fine tuning the parameters and approach, but I have not spent much time on fine tuning each test.

One of the tests differs significally from the others in that it does only use codes from some parts of the images. I belive this is a good approach to use in future studies of LBP on this and similar data sets, since the other approaches often end drowning the usable data in large amount of unusable data (and we get the same problem as illustrated in section 4.2.2). More approaches of this kind are discussed in chapter 7.

In the result section of tests with basic LBP (section 6.4) I list some more details than in the other sections.

6.2 Parameters and approach

The distance measure is in all cases Mahalanobis distance and the feature selection method is sequential forward selection.

In order to make sure the results are not overestimates without reducing the classification rate unneccesary the following process is followed: For each observation we first do feature selection based on all the other observations. Then we train based on all the other observations and the features chosen. We finaly classify the observation left out. Including the observation we wish to test as a part of the data used for feature selection will overestimate the classification accuracy [5].

Some tests use a subset containing of equal number of patients from each class other use all patients patients. This will be indicated under the descriptions of the individual tests.

The images have been grouped in different ways (se the "The test and parameter" section for each test for details"), in most of the tests all of the images for each patient are treated as one large cell surface area. This is done by taking the sum of the frequency for each LBP code in all images, since all cells are given the same weight this will favor the smaller cells since the frequencies are normalized.

Both a Matlab implementation of LBP made by Marko Heikkilä and Timo Ahonen downloaded from `http://www.ee.oulu.fi/mvg/page/lbp_matlab` and my implementation is used for the tests. The description of the tests indicate if the Heikkilä and Ahonen implementation is used. This implementation does the same work as mine, but it is included to show that the poor classification rates is not only present for my implementation.

6.3 Short summary of the results

Test name	**CCR**	**Specificity**	**Sensitivity**	**See section**
Standard1	0.6071	0.1111	0.8421	6.4
Standard2	0.6310	0.2593	0.8070	6.4
Standard3	0.5185	0.5556	0.4815	6.4
Large	0.5741	0.4074	0.7407	6.5
Size	0.6786	0.4074	0.8070	6.6
Gray level	0.6429	0.2593	0.8246	6.7
Max	0.5000	0.4815	0.5185	6.8
Variance	0.3889	0.3333	0.4444	6.8

Please note that some of the tests use all patients while other use an equal number of patients from both the good and bad prognosis class, see the sections listed for details. The results in section 6.9 is not listed in this table, since it contains so many tests compared to the other sections.

6.4 Standard LBP

6.4.1 Motivation and weaknesses

This is the most usual way to use LBP codes, and should therefor be tested. It is a simple approach, but should still be a good choice in many cases.

It does not have any weaknesses compared to the other methods tested, and is a good choice in the general case. The other methods do however have different strengths that this methods lacks.

6.4.2 The test and parameters

The goal of this test is to see how LBP in its original version performes. We only use the rotation invariant version since there is no way of knowing which way the cells are rotated. The radius used is 1 and the number of surrounding pixels are 8. There is not used any interpolation, the 8-neighbourhood is used.

Any inconsistencies between the implementations are due to differences in handling of the edges of the images. The LBP frequencies used is the average of the frequencies for all the images for each patient.

Test "Standard1"

Heikkilä and Ahonen implementation. All patients are used.

Test "Standard2"

My implementation. All patients are used.

Test "Standard3"

My implementation, an equal number of patients from each of the two groups were selected.

The LBP used the highest number of times were: 5 (binary: 101, 20 times), 7 (binary: 111, 13 times), 9 (binary:1001 10 times), 31 (binary:11111, 33 times) and 127 (binary: 1111111, 20 times). Histograms for these features can be found in Figure D.1 to D.5 in the appendix. The histograms show that the average value is almost the same for both classes for all codes, and shows no clear signs of good class discrimination potensial.

On average 2.6 features were used.

6.4.3 The results

Test	CCR	Specificity	Sensitivity	F. neg.	F. pos.	T. neg.	T. pos.
Standard1	0.6071	0.1111	0.8421	9	24	3	48
Standard2	0.6310	0.2593	0.8070	11	20	7	46
Standard3	0.5185	0.5556	0.4815	14	12	15	13

6.5 LBP with other radiuses and number of points

6.5.1 Motivation and weaknesses

In the previous test we only looked at the standard rotation invariant LBP which uses a radius of 1 and 8 surrounding points.

The reason for not using more than 8 points is that this gives rapidly increasing number of possible values, using 16 points gives $2^{16} = 65536$ non rotation invariant codes. The radius does not have any effect on the number of codes.

The increase in number of features complicates the feature selection, and also increases computation times in other stages.

6.5.2 The test and parameters

No grouping of the features is done in these tests. The number of patients from bad and good prognosis group is equal.

Test "Large"

Large collection uses all combinations of the following parameters:

points 4 or 8 surrounding points

radius radiuses of 1, 2, 4 and 8 pixels

interpolation nearest interpolation

cenInterpol nearest interpolation

calculation normal or forgiving(35)

movePattern flat(1), meaning that all pixels in the cell nuclei is used

6.5.3 The results

Test	CCR	Specificity	Sensitivity	F. neg.	F. pos.	T. neg.	T. pos.
Large	0.5741	0.4074	0.7407	7	16	11	20

6.6 Grouping based on cell nuclei size

6.6.1 Motivation and weaknesses

The main reason for grouping based on size is if we feel that the textures is different in the nuclei of different sizes. We do not have any concrete evidence of this, but it sounds plausible. Other measures might be better at discriminating between nuclei with different texture sizes, the areas of dark and light areas for example might be a good guess. Even better ways of estimating the textures scale might be possible to develop by those who know more about the biology and has more familiarity with what happens to the nuclei

in the image production process than me. So using the nuclei size admittedly probably is not an optimal approach, but it might be a step towards the right path.

Note that even though the nuclei size itself probably contains information that might be usefull in discriminating the classes, we do not want to mix in this information. The cell sizes is for this reason only used to group the features, not to guide the decision.

6.6.2 The test and parameters

We here group the LBP-codes based on the size of nucleus. This way we get N groups for each patient. These could be counted as different observations, but instead we try using each group as a seperate feature.

So instead of having F features we get N*F features, and one observation per patient per feature.

Since the cell sizes vary not all groups exist for all patients. This is in this test handled by removing the feature if it does not exist for all patients. This is not an optimal solution, but other solutions will give problems with index consistency and make it much easier to do errors during feature selection and classification.

Since there has been some uncertainty whether all the small cells actually are cancer cells, cells below the lowest thresholds are not used.

The thresholds for grouping is every thousand from 1000 to 9000. The LBP implementation used is Heikkilä and Ahonens. The 36 rotation invariant LBP codes for a radius of 1 and 8 points is used. All patients have been used.

6.6.3 The results

Test	CCR	Specificity	Sensitivity	F. neg.	F. pos.	T. neg.	T. pos.
Size	0.6786	0.4074	0.8070	11	16	11	46

6.7 Grouping based on average gray level

6.7.1 Motivation and weaknesses

The motivation for grouping based on gray level is based on the same logic as for grouping based on nuclei size (see section 6.6). The differences in color might indicate nuclei in different stages, pressure etc.

6.7.2 The test and parameters

This test is for most part the same as “Grouping based on cell nucleus size” (section 6.6). The only difference is that we group based on the average gray level in the image instead og the size of the nucleus.

My implementation and all patients are used, the LBP codes used are the same as the ones in section 6.5. The LBP codes used is the same as the ones used in section 6.5.

The thresholds used are: 200, 300, 400, 500, 600 and 700

6.7.3 The results

Test	CCR	Specificity	Sensitivity	F. neg.	F. pos.	T. neg.	T. pos.
Gray level	0.6429	0.2593	0.8246	10	20	7	47

6.8 Using maximum value or variance

6.8.1 Motivation and weaknesses

The tests performed up to this point have used the mean either for all cells or within the groups. This has not shown promising results. The weakness of using the mean value is that it might be based on too much data, and the interestion data drowns. In a first effort to counter that effect the variance or maximum value can be used. These also use all the data to be calculated, but will both put more weight on the extreme observations. Minimum can not so easily be used since this will give many zeros and problems calculating covariance matrixes and is therefor left out.

The weakness in using the maximum value and variance is that we still rely on all the data. The approaches are also rough, and does not take any consideration what might be special for the data set.

6.8.2 The test and parameters

Both tests where run on the collection on LBP codes from section 6.5. An equal number of patients from the good and bad prognosis class was used.

6.8.3 The results

Test	CCR	Specificity	Sensitivity	F. neg.	F. pos.	T. neg.	T. pos.
Max	0.5000	0.4815	0.5185	13	14	13	14
Variance	0.3889	0.3333	0.4444	15	18	9	12

6.9 Only using codes from dark regions

6.9.1 Motivation and weaknesses

The reason for looking at what looks like dark regions in the images (but which is the pixels with the highest pixel values) instead of light regions is based on the data set.

6.9.2 The test and parameters

All the tests are performed on a data set with equal amount of good and bad prognosis patients. Marko Heikkilä and Timo Ahonens implementation is used since I then just needed to do small alteration to my Matlab code. The code used can be found in code listing 15 in the appendix. This code looks at the max pixel values for each column of the cell image array, and chooses the Nth largest as a threshold. LBP codes is then calculated for all positions with center pixel larger than the threshold.

5th largest

0.52% of the LBP codes was used.

6th largest

0.67% of the LBP codes was used.

7th largest

0.84% of the LBP codes was used. Was also run without balancing the number of patients from each group. Feature for LBP code 0 was most used with 46 times in the balanced data set, but in the unbalanced data set it was only used 19 times (even though the total number of possible times were higher). The reason for this becomes clear when we look at the histograms of the distribution of LBP code 0 for the two patient groups in the two cases (see Figure D.6 and D.7 in the appendix).

8th largest

1.01% of the LBP codes was used. This was also run without balancing the number of patients from each group, this had the same effect as for 7th largest and 9th largest.

9th largest

1.21% of the LBP codes was used. Was also run without balancing the number of patients from each group. Same effect as on the two previous, see Figure D.8 and D.9 in the appendix to see the difference in the distribution of LBP code 7.

10th largest

1.41% of the LBP codes was used.

11th largest

1.64% of the LBP codes was used.

6.9.3 The results

Test	CCR	Specificity	Sensitivity	F. neg.	F. pos.	T. neg.	T. pos.
5th largest	0.5926	0.4074	0.7778	6	16	11	21
6th largest	0.5556	0.4074	0.7037	8	16	11	19
7th largest	0.6481	0.3704	0.9259	2	17	10	25
7th unbal.	0.5357	0.3333	0.6316	21	18	9	36
8th largest	0.6111	0.5185	0.7037	8	13	14	19
8th unbal.	0.5238	0.5926	0.4912	29	11	16	28
9th largest	0.6481	0.5185	0.7778	6	13	14	21
9th unbal.	0.5119	0.3333	0.5965	23	18	9	34
10th largest	0.5000	0.1852	0.8148	5	22	5	22
11th largest	0.5370	0.1852	0.8889	3	22	5	24

Chapter 7

Other possibilities

7.1 Introduction

The possibilites listed in this chapter have to a varied extent been tested and have code for computing it. In the chapter about the application in the appendix I also list some other suggestions to new features that requiers altering the LBP implementation (see section A.9.2). There is a lot of other possibilities than the ones already tested and the ones listed here. What should be tried depends on the data set. The possibilites listed in this chapter are meant for this data set.

7.2 General aspects

We have already tried to group the observations by size, but by mixing this with different radiuses we could get a degree of scale invariance. This does however require that we compare what now is seperate features to each other.

More fine tuning on the fuzzyness could also be tried. The images should also be considered grouped in other ways to make sure we test features that have been calculated in as simular environments as possible.

We should be looking at only parts of the nuclei in order to get as powerful features as possible.

7.3 Only comparing to good nuclei from good patients

Selecting one or more "ideal" nuclei in each class and only comparing to these might be a good approach in some cases.

7.4 Only look at cells or areas with high or low variance

The theory behind only looking at cells or areas with high or low variance is that these might contain more diagnostic information.

We could also try to group the features based on the variance, but this will probably not do much as long as we do not combine it with something else.

7.5 Creative interpolation functions

The implementation used to calculate the LBP-codes can use a custom function to interpolate the values of the surrounding pixels, and also another custom function to calculate the center value. These functions do not need to just interpolate the pixel values, we could also calculate other things to use as the value, for example the local maximum or minimum. This could typically be combined with a spacing move pattern and larger radiuses

In this way we could for example search for patterns of high and low variance areas. Or patterns in the distance from each surrounding points to a pixel with a given color.

7.6 Grouping based on where we are in the image

The LBP codes could be grouped after the gray level of the center pixel, a version of this has been done by Nielsen et al. [9]. We could also group based the values of the coordinates in order to for example assign different weights at a later point.

Chapter 8

Conclusion

LBP is an simple and intuitive idea that can be molded into almost any measure we like. It has great potential, but at the current time it lacks some more advanced software in order to be an easy solution to hard problems.

The results seen in the tests all indicate that LBP does not discriminate the classes with any degree that suggests it should be used for this purpouse on this data set.

This does not mean that it might not be usable in other applications of the same kind. It might also be usable in an other form on the same data set, if we know more what we are looking for or test other features. We need to either find a new approach or have some new insight to where in the nuclei we should look for information. LBP might then be a very powerful technique.

It does seem clear that at the current point in time efforts should be put into other methods on this data set.

The experiments performed have used seperate training and test data sets. Given the limited number of patient cases aviable, the sizes of the data sets has been small. This implies a high variance, which might have resulted in potentially good features being lost. Only a larger data set can remedy this.

Appendix A

More about the application

We will here look at the basic functionality of the application. For a quick introduction on how to use it see A.2. Some of the functions will be described in A.7 and a short introduction to how the code is organized can be found in A.11.

There exsist much more functionality and options than are described here. Some parts of the software are not used at all in this thesis since this has been implemented in Matlab. A short description of these parts can be found in A.8.

The main objective of the thesis is the results not the code producing it, and therefor, nither the code, comments in the code or the documentation is of the standard I usually want it to have before releasing it. The code may however be of interest for others and therefore I have chosen to include it even though is see weaknesses in it an in many ways would want to fine tune it before showing it to anyone else. So use it at own risk. Some functions will delete files on the filesystem, so be carefull and do not run the code without being sure what it does.

A.1 Requirements

The application need Java 5.0 or higher with JAI (Java Advanced Imaging)

`http://java.sun.com/javase/technologies/desktop/media/jai/`

installed. In order to have PNG-support javapng

`http://code.google.com/p/javapng/`

is also required. Note that none of the two are included with the code aviable online.

JAI needs to be installed on the computer the application will be run on, javapng can be added to the jarfile or on the classpath. In order to make an jar file also

A.2 Easy usage

The application can be run from the command line by running the LBP.jar file itself. This will run the main function made for use at RadiumHospitalet:

```
java -jar LBP.jar imageFolder outFolder pairs
```

imageFolder Path to a filelist containing all images we want to use, see section A.3.1.

pairs Path to a specially formatted file that describes what to do with the images, see section A.3.2.

outfolder A folder to write all calulated data to, if the folder does not exist it will be created. All existing files in the folder will be overwritten.

Running the command in this way calls the function readAndSave (see section A.7.1). Existing files will be overwritten and grouping of images will be performed.

For information on the format of the images list file see section A.3.1. Information about the format of the pairs parameter file can be found in A.3.2.

Information on what is returned can be found in section A.4.

Example

```
java -jar -Xmx1000m LBP.jar /Users/howie/Desktop/temp/link/L23/ \
/Users/howie/Desktop/skalSlettes /Users/howie/Desktop/pairs \
onlyRoted:no readyMasks:/Users/howie/Desktop/allRegLBPDone/masks/
```

Here we have added a parameter to the Java viritual machine that allows the application to use more memory (1000 mega bytes in this case). The memory usage will depend on the parameters supplied and the size and number of images.

A.3 Indata

A.3.1 The file list

The format in the file lists is as follows:

```
filePathImage class filePathMask group
```

filePathImage The file path to the image

class The known class of the image (if it is it unknown, a class still has to be provided so just use "unknown")

filePathMask Path to an image to use as mask, this is optional and can be omitted

group A string that is the same for all images of the same group (NOTE: images of the same group needs to be listed together, with no other images separating them)

Note that the separator is a space character so the paths and names can not contain spaces (if this should be an issue, it is not a big job to alter this behavior). Lines starting with # are treated as comments.

A discription of the organization of the data set can be found in section 2.3.

Example:

```
/L23/046/Images/L23-046_0_cell.tif good /temp/masks/046/0.png 24
/L23/047/Images/L23-046_0_cell.tif good /temp/masks/046/0.png 25
```

A.3.2 The "pairs" file

The "pairs" file is a file describing which pairs of parameters that should be supplied to the LBP function. In addition we specify some details about how it should be calculated. This is the format of each line in the pairs list:

```
function(paramsToFunc) interpolation cenInterpol calculation movePattern
```

function Name of the function (only "LBP" is supported)

paramsToFunc The parameters to the function, comma separated

interpolation Specifies which interpolation function to use to interpolate the surrounding points ("bilinear" and "nearest" is supported)

calculation Gives the possiblity to choose alternative methods of calculating the LBP code use "normal" for regular LBP

cenInterpol Added in order to make some extended versions of LBP, use "normal" for regular LBP

movePattern This is a combined mask and weighting function. Use flat(1) to use all pixels and weight them equally.

The pipe character can be used as an "or" operator. All possible combinations will be used.

Example:

```
LBP(8,1) nearest nearest normal flat(1)
LBP(4|16|64,1|5|10) bilinear nearest normal flat(1)
```

A.3.3 Supported image formats

The application should suppport JPEG, TIFF and PNG images. The default behavior at the current point in time is to convert all color images into grayscale. 10 bit TIFFs can be read as they are (internally the pixeldata is saved as 32 bit integers, also for grayscale images).

A.3.4 Supported mask formats

Masks are images in one of the supported image formats (see A.3.3) that have values 0 for pixels that are not part of the object of interest and positive vaules in all pixels that are of interest (no weighting based on pixelvalues is done at this point).

A.4 Out data

All out data are written as plain text. There are many reasons for doing it in another way, but since I did not know what was needed I did it the easiest (and most versatile) way.

A.4.1 The analysis oriented format

This format has the strength that it does not need as many files and folders on the hard drive as the complete format (next section). It does also require less memory while calculating since the data is written to disc after each file is read.

It does however not save anything else than patient numbers, parameter combinations and relative frequencies.

A.4.2 The complete format

One folder will be created and six files containing the file list and pairs list are written in an easy readable format. Data saved in this fromat can be read from disc by the readFromFolder-method (see section A.7.2)

In addition to this one folder will be created for each image (at this point all images are saved sepparately, also the ones that are grouped). In each of theese subfolders there will be written four files for each possible combination in the pairs file described in section A.3.2. The four files are:

allcodesNNN.txt Contains all the LBP-codes (will be empty unless we set the full parameter, see A.7.1)

allrcodesNNN.txt : contains all the rotated LBP-codes (will be empty unless we set the full parameter, see A.7.1)

codesNNN.txt : contains a frequency table for all the LBP-codes

rcodesNNN.txt : contains a frequency table for all the rotated LBP-codes

A.5 Special features of this implementation

This implementation has been specialized for use at Radiumhospitalet, but the specialized parts are made as a translator in the package translator.radiumHospitalet.

In addition to beeing specialized for use at Radiumhospitalet there are some additions and possibilites for custimization.

A.6 Threads and calculation speed

The application does per defualt use the same the number of threads as the number of processors on the system for calculation of LBP-codes, and should therefore use close to all available processing power on all CPU-cores. The splitting into threads is done on patient-basis.

Note that generating mask images is done by only one thread. The reason for this is that generating mask images only has to be done once, and takes no more than 20 minutes on a standard laptop for all images for 120 patients. These masks are then saved and can be used when running LBP at a later point, see A.2.

Creating of filelists is done by one thread and for every run. This is not neccesary, but since the time usage is so small compared to running LBP (about 5% of the runtime for the easiest LBP-calculations), it is done every time to limit the number of parameters and possible error sources (since this process will check that all image files exsist before starting the much more time consuming LBP-calculations).

Also note that since each image can be analysed by itself, the task can also be split to multiple systems by simply splitting the data set.

Some sample runtimes can be found in A.10

A.7 Methods

The application is primarily made to be controlled by a Java-class in order to increase the speed and flexibility. The main functions are all located in core.Fileopearions:

A.7.1 readAndSave

This is the most important function

```
public static ResultCollection readAndSave(String images, String pairs, String outFol
boolean full, boolean overwrite, boolean useGroups)
```

images Path to a filelist containing all images we want to use, see section A.3.1.

pairs Path to a specially formatted file that describes what to do with the images, see section A.3.2.

outFolder A folder to write all calulated data to.

full Indicates if all codes should be saved not just the counts.

overwrite Indicates if overwriting existing data is allowed.

useGroups Indicates if data about grouping in the pairs-file should be applied, see section A.7.3.

It returns an ResultCollection-object which contains all the same data that is written to the outfolder (the supplied full parameter affects the .

There also exist some methods that overloads this one, but this is the most general one

A.7.2 readFromFolder

This funtion might be usefull if you want to manipulated saved data

```
public static ResultCollection readFromFolder(String folderPath,boolean useGroups)
```

folderPath Path to the folder to read from.

useGroups Indicates if data about grouping in the pairs-file should be applied, see section A.7.3.

This function reads in the data from the filefolder and returns an (in practice) equal ResultCollection object as the readAndSave (see A.7.1) function that wrote the files returns.

A.7.3 Grouping of images

When images are grouped the collected data is combined, which lead to lower memory usage and smaller files if saved on disc.

IMPORTANT: Images of the same group can not have other images seperating them in the list in the file list A.3.1

A.8 Other functionality

A.8.1 Feature selection and Classification

Both Sequential Forward Selection and Floating search has been implemented, but only Sequential Forward Selection is completed.

A.9 Possible extensions

I see a lot of more things I would like to implement that I feel would be appropriate as a part of this software. A lot of them have been in mind while implementing, and should be easy to implement at a later point. Some of the features also have some code that has been commented out or put in to functions that have no good use on their own, examples include:

- Support for extending LBP in different ways (partly implemented, and usable to some extent)
- Support for other other functions than LBP
- Support for using the full lists of codes, not only the counts

This software was meant to be a proof of consept, and this is why I have not removed the parts that are not finished yet. But in addition to this there is also some features I haven't had to much in mind:

A.9.1 Scale invariance

Scale invariance is an important part for many uses of texture analysis. This is not supported by the software at this time. It can be implemented by not only looking at data within each LBP-parameter combination (its the R-that is varing with the scale), but also look at data for different LBP-parameters. This is not implemented since it is not relevant for the intended primary useage of this application, since the scale on the images should not change.

A.9.2 More features

A lot of extended or altered versions of LBP could be of interest:

Linear/Square LBP

Instead of looking at surrounding values in a circle around the center we could look at pixels orignized in some other pattern that might be more usefull considering the texture in the image. This would however give problems with calculation rotation invariant codes, since the approach now used relies on them beeing in a circle.

Local Decimal/Hexadecimal Pattern

Instead of only looking at if the surrounding pixels are darker or not we, could do some kind of grouping. This breaks with the basic idea (and name) of LBP,

Gray level grouped

We could group the codes based on the gray level in the center pixel.

Adding support for other kinds of features than LBP-based is also a possibility. The data set we are working with will often have some kind of pattern we want to optimize the function to find.

A.9.3 Make it more automatic

This implementation requires the user to specify which parameters to use. For easy usage in new applications automatic selection based on some image measures might be an possibility.

The application could also have other main methods that support other dataformats than the one used at Radiumhospitalet.

A.10 Example calculation times

Some calculation times:

```
Pairs: LBP(8,1) nearest nearest normal flat(1)
java -jar -Xmx1000m LBP.jar ../../L23/ /Users/howie/Desktop/allRegLBP  \
 /Users/howie/Desktop/miniset/pairs onlyRoted:yes
```

```
Creating mask images finished in 16m10s
Creating file lists finished in 1m26s
Calculating LBPs finished in 19m11s
Fininshed all tasks in 36m48s


Pairs: LBP(8,1) nearest nearest fuzzy(3,5)|fuzzy(2,2) flat(1)
java -jar -Xmx1000m LBP.jar ../../L23/ /Users/howie/Desktop/allFuzzyLBP  \
/Users/howie/Desktop/miniset/pairs onlyRoted:yes
Creating mask images finished in 15m35s
Creating file lists finished in 1m27s
Calculating LBPs finished in 21m37s
Fininshed all tasks in 38m40s


Pairs:LBP(8,1) nearest nearest normal maskCenter(circle,25\%,25\%,2,1,2)|
maskCenter(circle,60\%,60\%,2,1,2)
java -jar -Xmx1000m LBP.jar ../../L23/ /Users/howie/Desktop/allCenterLBP \
/Users/howie/Desktop/miniset/pairs onlyRoted:yes \
readyMasks:/Users/howie/Desktop/allFuzzyLBP/masks/
Creating mask images finished in 0m0s
Creating file lists finished in 1m16s
Calculating LBPs finished in 19m18s
Fininshed all tasks in 20m35s
```

A.11 Packages

core Contains the most central classes in the Application

image Contains classes representing image objects

interpolation Classes for interpolation of image points

lbp LBP-specific classes

movepattern Contains classes differnt kinds of move patterns

result Cointans classes for storing and filtering results

selection Classes needed for feature selection

statistics Classes for calculating and saving statistical data

translators Classes for finetuning the applicaiton to the environment

Appendix B

The Python version of the software

B.1 Pyhton and Java

I started to program in Python, for many reasons:

- A pretty new language to me, which gave it a great learning potential.
- Python is often used to prototype applications, so I wanted to try to use it for that purpouse.
- The language is used by several of the huge software developers, and is gaining popularity.
- As far as I can see no public LBP-implementation in Python exists.
- Not as strict as Java, lets you do what you want with less code.

The second implementation was done in Java:

- No Java implementation existed either.
- Java is pretty fast.
- Java is in very common use.
- Gives a more organized code (at least for me).
- My primary programming language.

In an attempt to speed up the Python implementation I tried using Jython. Jython is a Python implementation in Java, that makes it possible to run Python code in java as Java classes. Support for passing python arrays to Java code did not seem to be implemented yet, thus the attempt fail.

Appendix C

Matlab code

In this appendix all the custom Matlab-functions that are refrenced to should be listed. They are probably not of an impressive standard for experienced Matlab users, but they are included to make it easy to test my results and find flaws in my implementation and logic. I have tried to use the most intuitve code not the shortest one, mostly to make it easier for my self to understand at a later point what the code actually does.

The layout of the folder and files used is desctibed in section 2.

Also the Matlab code has grown big, especially the FeatureCollection class which is over 1400 lines. This is too much to list nicely in this document and also probably more code than most want to look at, so i just list the most important functions here. But the full source code can be found together with the rest of the code on `www.freso.net/lbp`.

Program 7 Matlab code for calculating information about size of the cell nuclei

```
function cellSizeDiagrams(obj, outFolder)
    goodIndexes = obj.booleanToIndexes(obj.goodPatientIndexes >0);
    badIndexes = obj.booleanToIndexes(obj.badPatientIndexes>0);
    allIndexes = [badIndexes goodIndexes];

    indexTypes = {allIndexes, goodIndexes, badIndexes};
    data = zeros(3, 6);
    for i=1:size(indexTypes,2)
        indexes = indexTypes{i};
        numPatients = size(indexes,2)
        numImages = sum(sum(obj.numberOfPixels(indexes, ...
            1:end) >0))
        sumNumPixels = sum(sum(obj.numberOfPixels(indexes, ...
            1:end)))
        averageNumPixels = sumNumPixels / numImages
        maxPixels = max(max(obj.numberOfPixels(indexes, 1:end)))
        minPixels = min(nonzeros(obj.numberOfPixels(indexes, 1:end)))
    end

    %create histograms of max nuclear sizes
    x = 5000:5000:40000;
    hist(max(obj.numberOfPixels(goodIndexes, 1:end)'), x)
    title('Max nuclei size for patients in the good prognosis class')
    saveas(gcf, [outFolder , 'maxKernelGood.png'])

    hist(max(obj.numberOfPixels(badIndexes, 1:end)'), x)
    title('Max nuclei size for patients in the bad prognosis class')
    saveas(gcf, [outFolder , 'maxKernelBad.png'])

    % We need to remember that that the array is filled with zeros
    badMean = sum(obj.numberOfPixels(badIndexes, 1:end)') ./ ...
        sum(obj.numberOfPixels(badIndexes, 1:end)' > 0);
    goodMean = sum(obj.numberOfPixels(goodIndexes, 1:end)') ./ ...
        sum(obj.numberOfPixels(goodIndexes, 1:end)' > 0);

    %create histograms of mean nuclear sizes
    x = 2000:500:6000;
    hist(goodMean, x)
    title('Mean nuclei size for patients in the good prognosis class')
    saveas(gcf, [outFolder , 'meanKernelGood.png'])

    hist(badMean, x)
    title('Mean nuclei size for patients in the bad prognosis class')
    saveas(gcf, [outFolder , 'meanKernelBad.png'])
end
```

Program 8 Matlab code for calculating information about the mean and variance gray levels for each image, part 1

```
function grayLevelDiagrams(obj, outFolder)
    %calcutaing standard deviations from the variance
    standardDeviation = sqrt(obj.grayLevelVariance);
    goodIndexes = obj.booleanToIndexes(obj.goodPatientIndexes >0);
    badIndexes = obj.booleanToIndexes(obj.badPatientIndexes>0);
    allIndexes = [badIndexes goodIndexes];

    indexTypes = {allIndexes, goodIndexes, badIndexes};

    for i=1:size(indexTypes,2)
        indexes = indexTypes{i};
        numPatients = size(indexes,2)
        numImages = sum(sum(obj.numberOfPixels(indexes, 1:end) >0));

        sumAverage = sum(sum(obj.grayLevelAverage(indexes, 1:end)));
        sumSD = sum(sum(standardDeviation(indexes, 1:end)));

        meanAverage = sumAverage / numImages
        meanSD = sumSD / numImages
    end

    %create histograms of variance in gray level
    %note that we can not use the built in mean function since
    %this will include the empty matrix elements

    %create histograms of average gray level
    x =150:20:470;
    hist(sum(obj.grayLevelAverage(goodIndexes, 1:end)') ./ ...
         sum(obj.grayLevelAverage(goodIndexes, 1:end)' > 0), x)
    title('Average gray level for patients in the good prognosis class')
    saveas(gcf, [outFolder , 'averageGood.png'])

    hist(sum(obj.grayLevelAverage(badIndexes, 1:end)') ./ ...
         sum(obj.grayLevelAverage(badIndexes, 1:end)' > 0), x)
    title('Average gray level for patients in the bad prognosis class')
    saveas(gcf, [outFolder , 'averageBad.png'])

% continues on next page
```

Program 9 Matlab code for calculating information about the mean and variance gray levels for each image, part 2

```
% continues from previous page

    x = 60:10:170;
    hist(sum(standardDeviation(goodIndexes, 1:end)') ./ ...
         sum(standardDeviation(goodIndexes, 1:end)' > 0), x)
    title('Mean standard deviation for patients in the good prognosis class')
    saveas(gcf, [outFolder , 'goodMean.png'])

    hist(sum(standardDeviation(badIndexes, 1:end)') ./ ...
         sum(standardDeviation(badIndexes, 1:end)' > 0), x)
    title('Mean standard deviation for patients in the bad prognosis class')
    saveas(gcf, [outFolder , 'badMean.png'])

    %create histograms of maximum variance in gray level
    x = 150:10:300;
    hist(max(standardDeviation(goodIndexes, 1:end)'), x)
    title(['Maximum standard deviation within cell nuclei for ', ...
        'patients in the good prognosis class'])
    saveas(gcf, [outFolder , 'goodMax.png'])

    hist(max(standardDeviation(badIndexes, 1:end)'), x)
    title(['Maximum standard deviation within a cell nuclei for ', ...
        'patients in the bad prognosis class'])
    saveas(gcf, [outFolder , 'badMax.png'])

    %create histograms of minimum variance in gray level
    %first replaze the empty elements in the matrix with "inf"
    sdevinf = standardDeviation;
    sdevinf(sdevinf < 1) = inf;

    x = 0:10:80;
    hist(min(sdevinf(goodIndexes, 1:end)'), x)
    title(['Minimum standard deviation within cell nuclei for ', ...
        'patients in the good prognosis class'])
    saveas(gcf, [outFolder , 'goodMin.png'])

    hist(min(sdevinf(badIndexes, 1:end)'), x)
    title(['Minimum standard deviation within cell nuclei for ', ...
        'patients in the bad prognosis class'])
    saveas(gcf, [outFolder , 'badMin.png'])
end
```

Program 10 Matlab code that does feature selection and calculates the results. This code is implemented as a script instead of function in order to easier manipulate the in and out data. The variables at the top of the script can be altered in order to chose how and what we should calculate.

```
recalculate = 'yes';
groupVar = 'greylevel';
distanceMeasure = 'mahalanobis';
collectionFolder = ['/Users/howie/Desktop/MasterThesis/LBPSamlinger', ...
    '/fullCollection/patientfiles'];
goodFile = '/Users/howie/Desktop/MasterThesis/matlab/goodandi.txt';
badFile= '/Users/howie/Desktop/MasterThesis/matlab/badandi.txt';
imageFolder = '/Users/howie/Desktop/MasterThesis/Diverse/link/L23';
useBasic = 'no';
balanceTrain = 'no';
balanceTest = 'no';
trainCutOff = 0.33; %in percent
nfold = 20;
opts = statset('display','iter');

if strcmp(groupVar, 'all')
    groupsToFeatures = 'no';
else
    groupsToFeatures = 'yes';
end

if strcmp(recalculate, 'yes')
    if exist('cellarr') && strcmp(useBasic, 'no')
       collection = FeatureCollection(collectionFolder, cellarr, ...
           groupVar, goodFile,badFile,groupsToFeatures, 'yes');
    else
       collection = FeatureCollection(collectionFolder, imageFolder, ...
           groupVar, goodFile,badFile, groupsToFeatures, 'yes');
       cellarr = collection.getStatisticsCellArray();
    end
end

if strcmp(useBasic, 'yes')
    features = collection.allBasicLBP;
else
    features = collection.allFeatures;
end
numLines = size(features,1);

% continues on next page
```

Program 11 Script for feature selection part 2.

```
% continues from pervious page

if strcmp(balanceTrain, 'yes')
    g = collection.goodPatientIndexes' .* (1:collection.numberOfPatients);
    g = g(g~=0);
    b = collection.badPatientIndexes' .* (1:collection.numberOfPatients);
    b = b(b~=0);
    numInTrain = collection.numberOfPatients * trainCutOff;
    trainLines = [];
    testLines = [];
    stop = floor(numInTrain/2);
    for i=1:stop
        trainLines = [trainLines g(i) b(i)];
    end

    if strcmp(balanceTest, 'yes')
        last = min(collection.numberOfGoodPatients,...
            collection.numberOfBadPatients);
        for i=(stop+1):last
            testLines = [testLines g(i) b(i)];
        end
    else
        for i=(stop+1):collection.numberOfGoodPatients
            testLines = [testLines g(i)];
        end
        for i=(stop+1):collection.numberOfBadPatients
            testLines = [testLines b(i)];
        end
    end

    prior.prob = [0.5 0.5];
    prior.group = {'good', 'bad'};
    trainLines = sort(trainLines);
    testLines = sort(testLines);
else
    prior.prob = [collection.numberOfGoodPatients/ ...
        collection.numberOfPatients collection.numberOfBadPatients/ ...
        collection.numberOfPatients];
    prior.group = {'good', 'bad'};

    trainLines = 1:trainCutOff*numLines;
    testLines = floor(trainCutOff*numLines)+1:numLines;
end

% continues on next page
```

Program 12 Script for feature selection part 3.

```
% continues from pervious page

% This function should be minimized to increase the correct classification
% rate
trainFunction = @(trainFeatures,trainClasses,testFeatures,testClasses)...
      (sum(~strcmp(testClasses,classify(testFeatures,...
      trainFeatures,trainClasses,distanceMeasure))));

% This function gives 1 for correct and 0 for wrong classification
testFunction = @(trainFeatures,trainClasses,testFeatures,testClasses)...
      ((strcmp(testClasses,classify(testFeatures,trainFeatures,...
      trainClasses,distanceMeasure))));

% This function is used to calculate the distance measure
altTest = @(trainFeatures,trainClasses,testFeatures,testClasses)...
      (CustomClassify(testFeatures,trainFeatures,...
      trainClasses,distanceMeasure, 'value', 'yes'));

allLines = [testLines trainLines];
numLines = size(allLines,2)
numTrueNeg = 0;
numTruePos = 0;
numFalsePos = 0;
numFalseNeg = 0;
timesSelected = zeros(size(indexes));

for lineNumber=1:numLines
    line = allLines(1,lineNumber)
    other = [1:lineNumber-1 lineNumber+1:numLines];
    trainFeatures = features(other,indexes);
    trainClasses = collection.allPatientNumbers(other,2);
    c = cvpartition(numLines-1,'leaveout');
    chosen = sequentialfs(trainFunction, trainFeatures, trainClasses, ...
        'cv',c);
    timesSelected = timesSelected + chosen;

    % do not use the prior probabilites since we use mahlanobis distance
    % and the prior probability is therfor not used anyway
    prior = [];
    indexes = [];

% continues on next page
```

Program 13 Script for feature selection part 4.

```
% continues from pervious page

    % Find features that can be used
    for feature=1:size(features,2)
        % Skip features where all are zero in either test or training, also
        % remove any feature that contains any NaN.
        if (sum(features(trainLines,feature),1) == 0) || ...
            (sum(features(testLines,feature),1) < 0.001) || ...
            isnan(sum(features(1:end,feature),1))
           continue;
        end
        indexes = [indexes feature];
    end

    % Update with the indexes chosen.
    trainFeatures = features(allLines(1, other),indexes(chosen));
    testFeatures = features(line,indexes(chosen));
    testClasses = collection.allPatientNumbers(line,2);

    result = testFunction(trainFeatures,trainClasses, ...
        testFeatures,testClasses);
    distances = altTest(trainFeatures,trainClasses,...
        testFeatures,testClasses);

    numTrueNeg = numTrueNeg + sum(collection.badPatientIndexes(line) ...
        .* result);
    numTruePos = numTruePos + sum(collection.goodPatientIndexes(line) ...
        .* result);
    numFalsePos = numFalsePos + sum(collection.badPatientIndexes(line) ...
        .* ~result);
    numFalseNeg = numFalseNeg + sum(collection.goodPatientIndexes(line) ...
        .* ~result);
end

% Print the results to screen
timesSelected
correctClassificationRate = (numTrueNeg + numTruePos) / (numFalsePos ...
    + numFalseNeg + numTrueNeg + numTruePos)
numTrueNeg
numTruePos
numFalsePos
numFalseNeg
specificity = numTrueNeg / (numTrueNeg + numFalsePos)
sensitivity = numTruePos / (numTruePos + numFalseNeg)
```

Program 14 Script for creating LBP histograms after running the script for feature selection.

```
outFolder = '/Users/howie/lbpHists/';

classForLine = collection.goodPatientIndexes(allLines);

nLines = size(allLines,2);
goodLines = [];
badLines = [];
for line=1:nLines
    if classForLine(line,1) % if good prognosis
        goodLines = [goodLines allLines(1,line)];
    else
        badLines = [badLines allLines(1,line)];
    end
end

featuresToPrint = [4 5 6 17 35];
lbpCodeForFeature = [5 7 9 31 127];

for featureNum=1:size(featuresToPrint,2)
    f = featuresToPrint(1, featureNum);
    code = num2str(lbpCodeForFeature(1, featureNum));

    hist(collection.allFeatures(goodLines, f))
    title(['Frequencies for LBP code ', code  ...
            ' for patients in the good prognosis class'])
    saveas(gcf, [outFolder , code , 'good', '.png'])

    hist(collection.allFeatures(badLines, f))
    title(['Frequencies for LBP code ', code  ...
            ' for patients in the bad prognosis class'])
    saveas(gcf, [outFolder , code , 'bad', '.png'])
end
```

Program 15 The part of the FeatureCollection class that chooses the LBP codes only from dark areas in the nuclei. The line altered during the tests is marked with a comment.

```
lbpImage = lbp(cell, 1, 8, mapping, 'image');

% Erode the mask to only get the codes from within the
% nuclei
eMask = imerode(mask, [1 1 1; 1 0 1; 1 1 1]);
lbpImage = int16(lbpImage) - (int16(~eMask(2:end-1,2:end-1) * 256));

% If told to do so only look at ceters with high pixel
% values
if (obj.basicOnlyHigh)
    msort = sort(max(cell));
    thres = msort(1, end-NN);% NN is the only number i varied

    hMask = cell > thres;

    percent = sum(sum(hMask)) / sum(sum(eMask));
    percentnum = percentnum + 1;
    percentsum = percentsum + percent;

    lbpImage = int16(lbpImage) .* ...
        int16(hMask(2:end-1,2:end-1));
end

tab = tabulate(lbpImage(find(lbpImage>= 0)));

dataLine = zeros(1,mapping.num);

for tabLine=1:size(tab,1)
    dataLine(1,tab(tabLine,1) + 1) = tab(tabLine,3) / 100;
end
```

Appendix D

LBP Histograms

The LBP histograms listed in this appendix are all created by running the code in listing 14. The x-axis is the frequencies of the LBP for each patient.

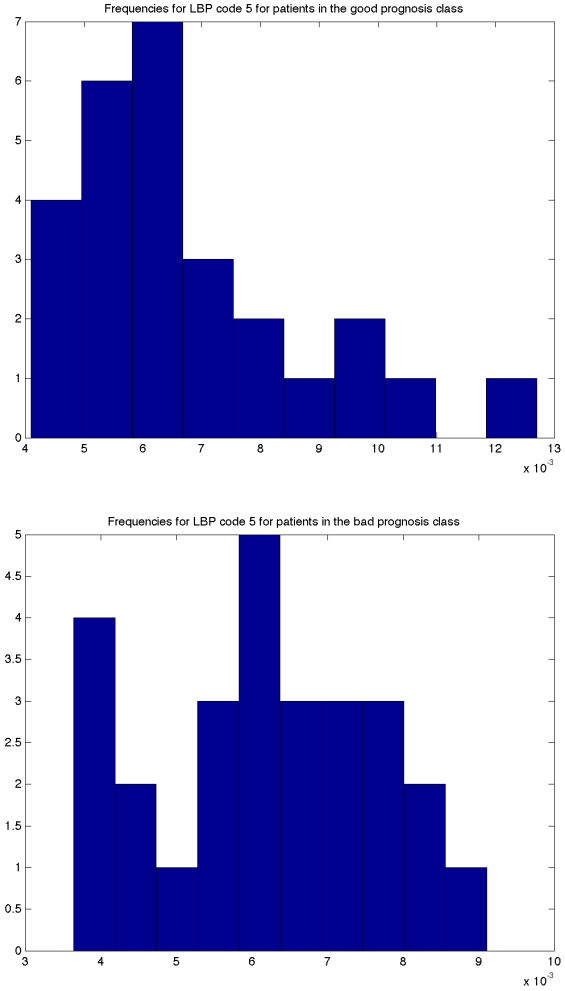

Figure D.1: Frequencies of LBP code 5 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

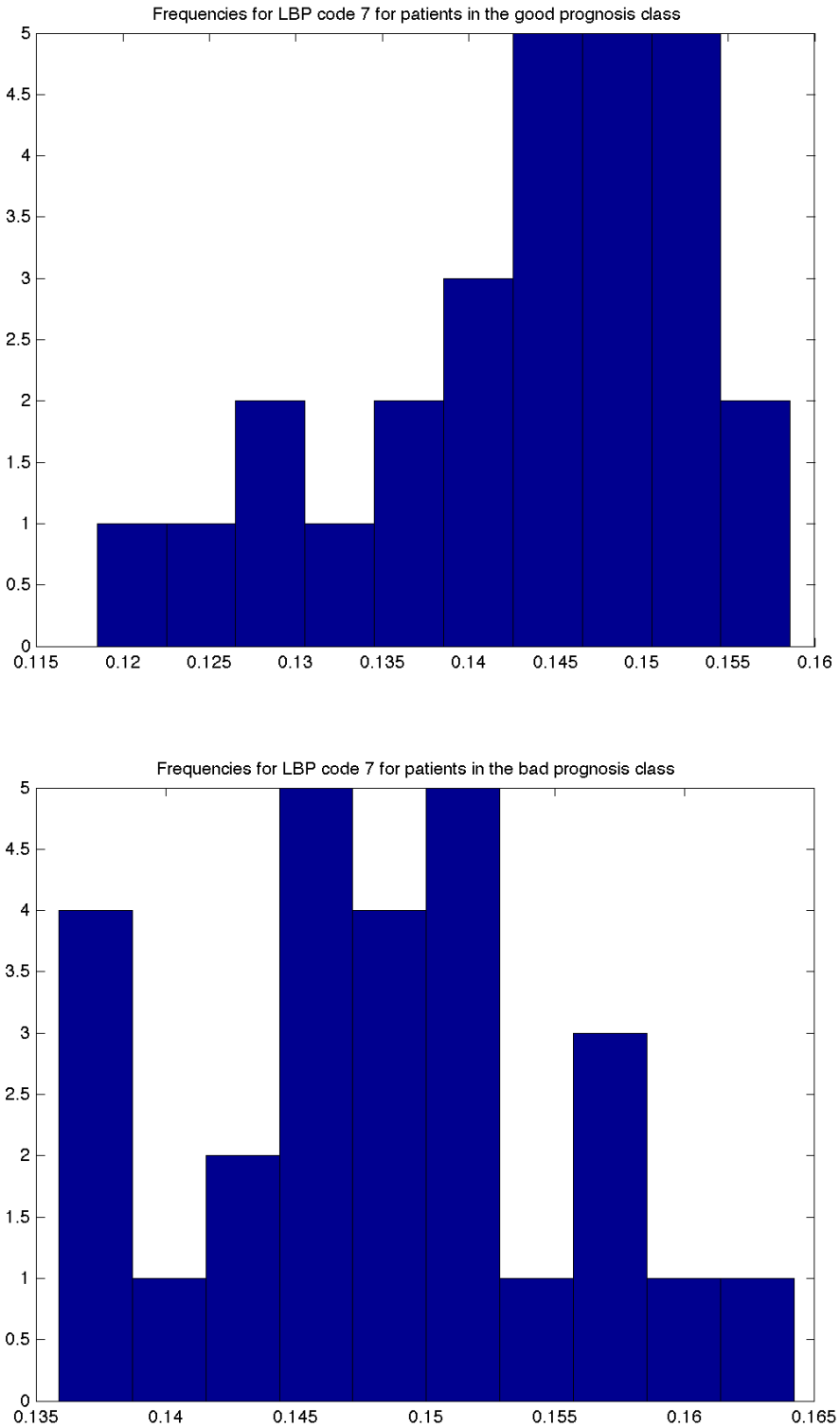

Figure D.2: Frequencies of LBP code 7 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

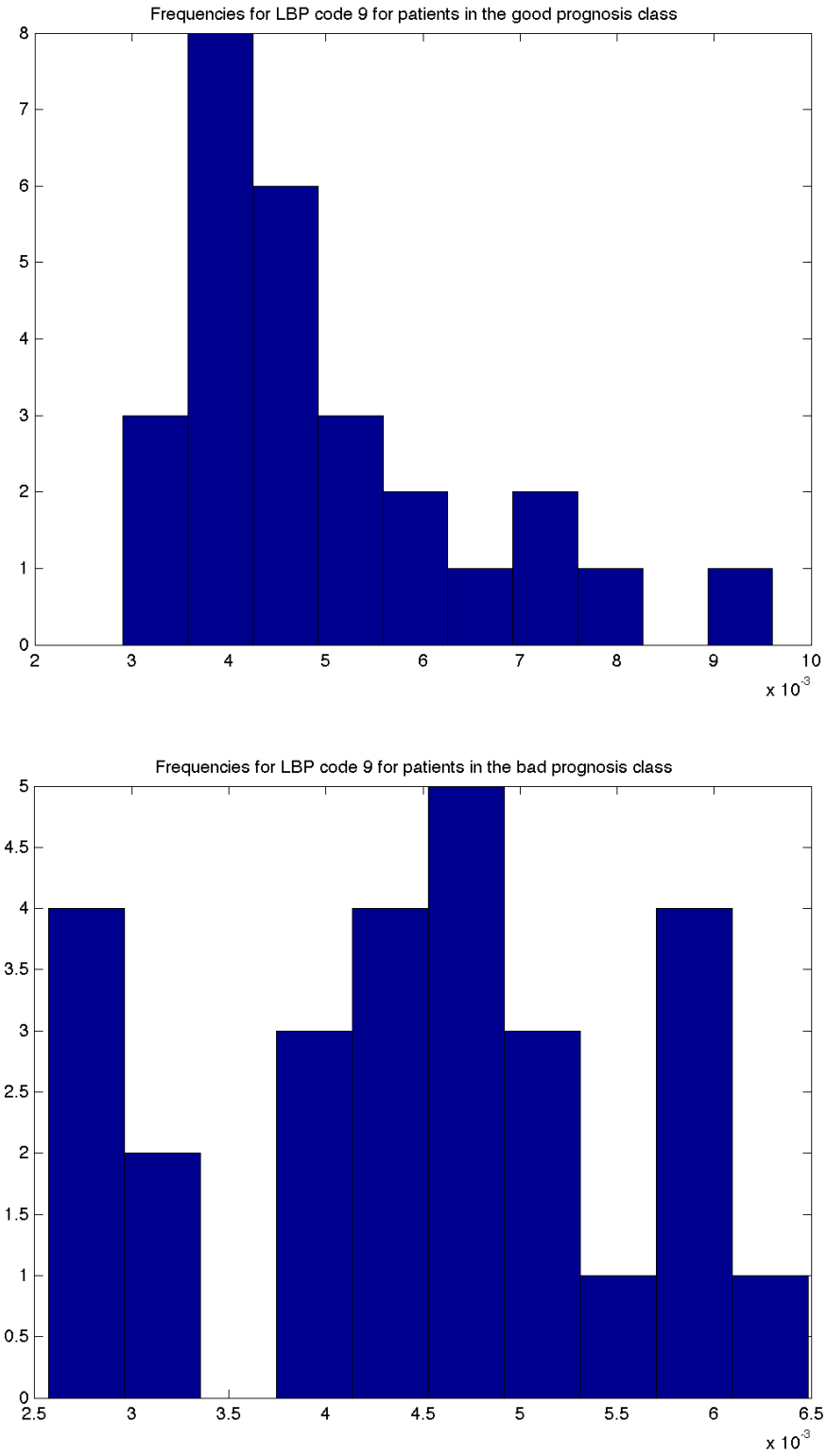

Figure D.3: Frequencies of LBP code 9 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

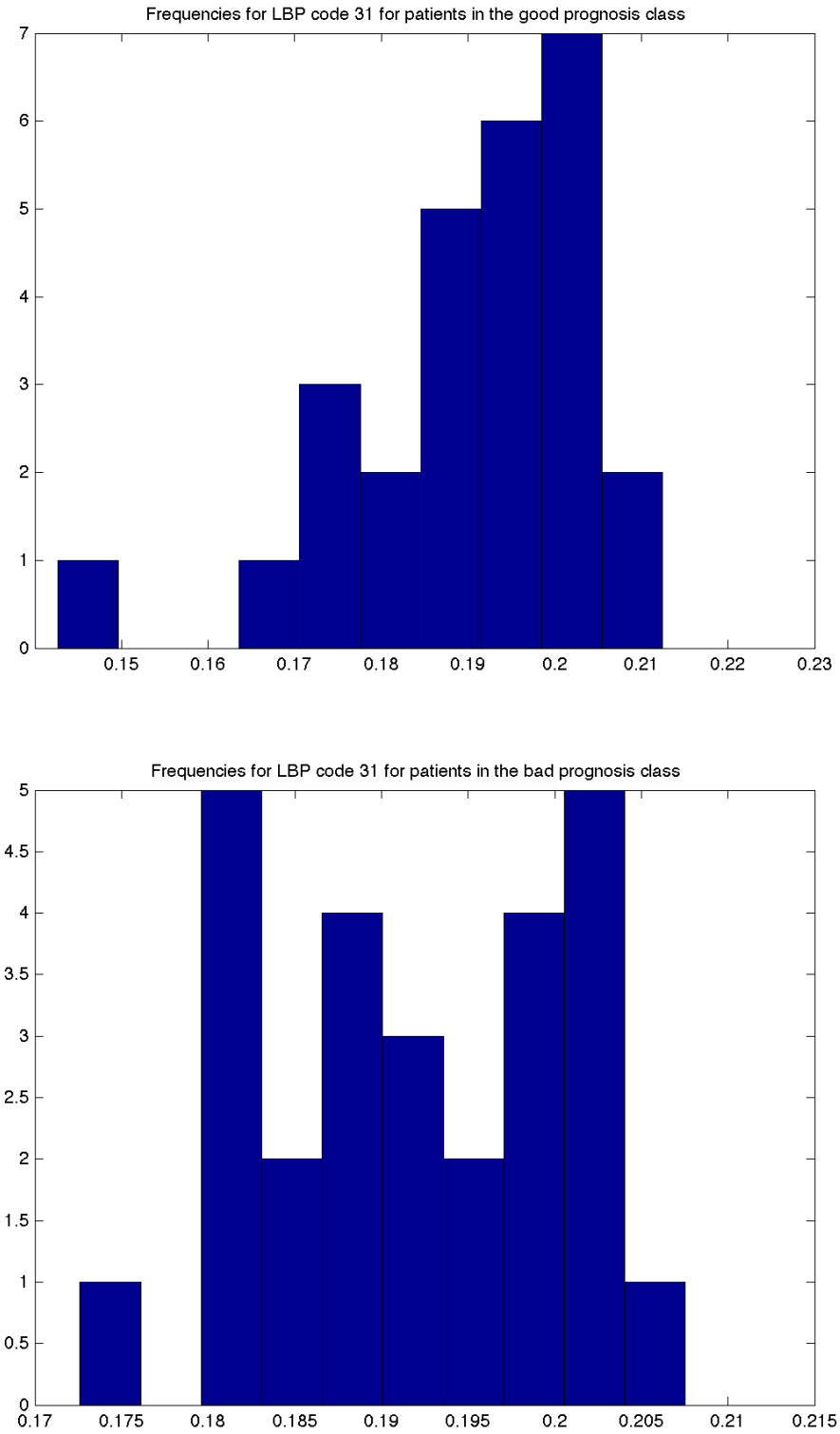

Figure D.4: Frequencies of LBP code 31 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

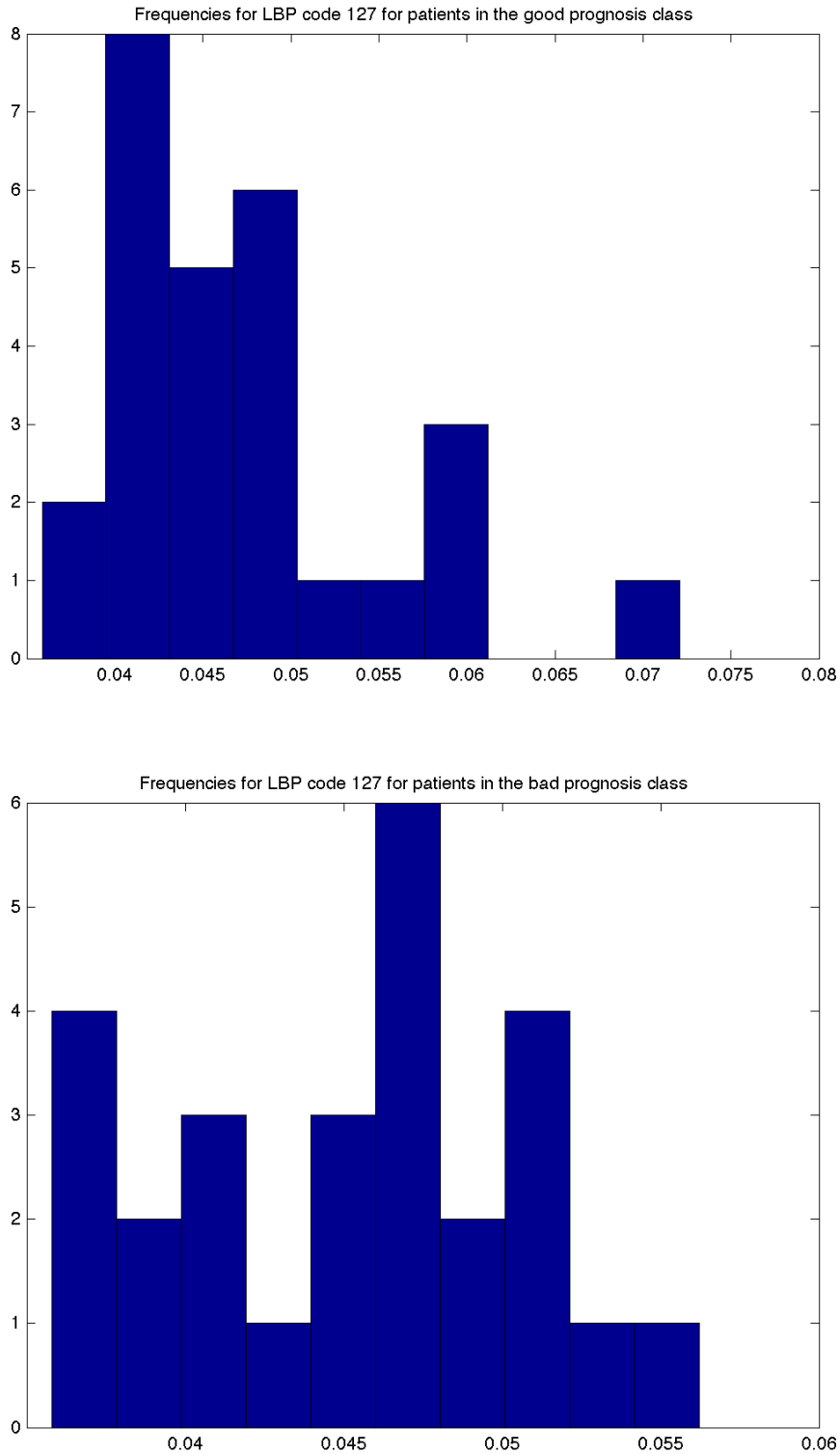

Figure D.5: Frequencies of LBP code 127 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom.

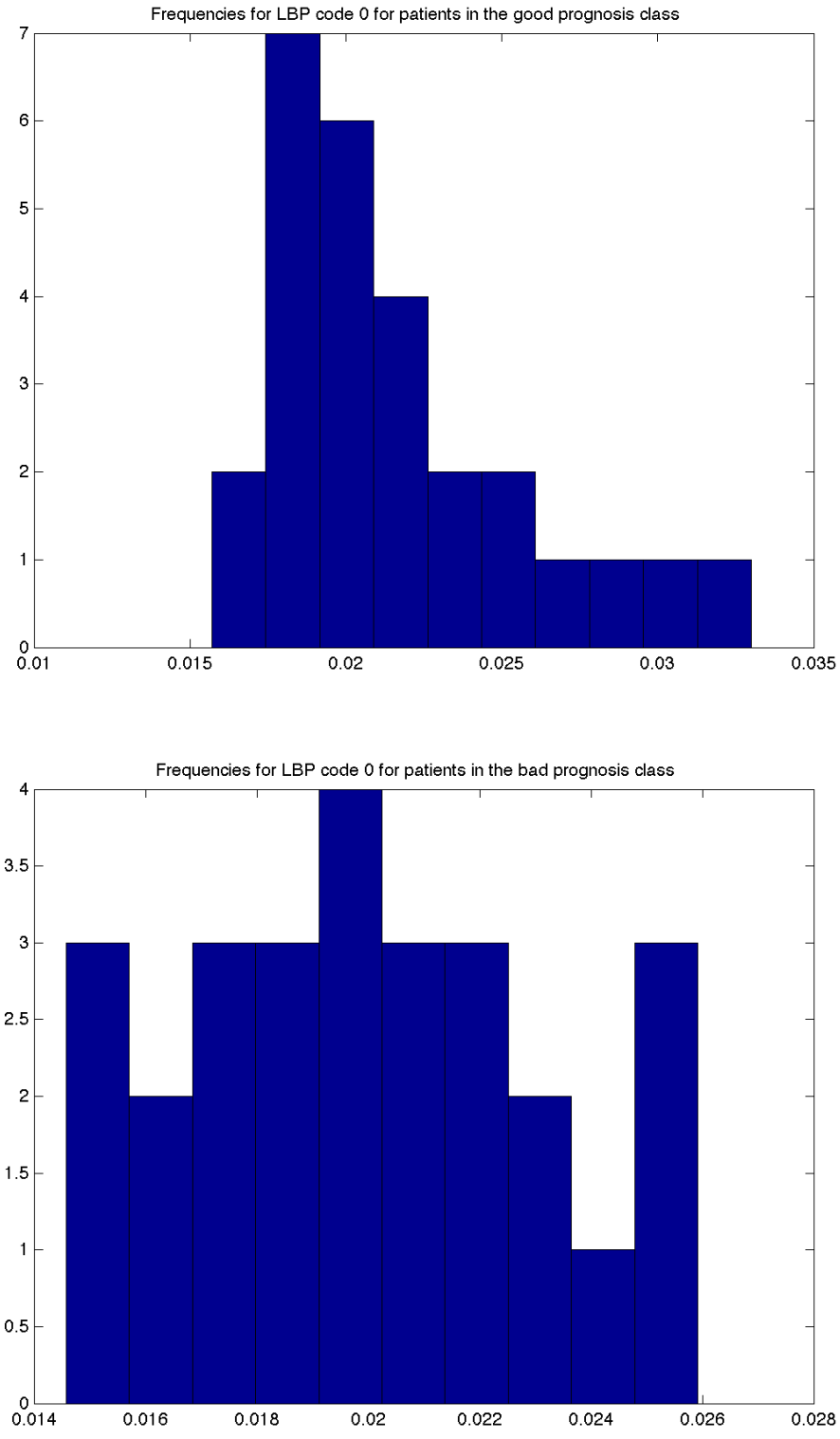

Figure D.6: Frequencies of LBP code 0 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom. Using 27 patients from each of the classes.

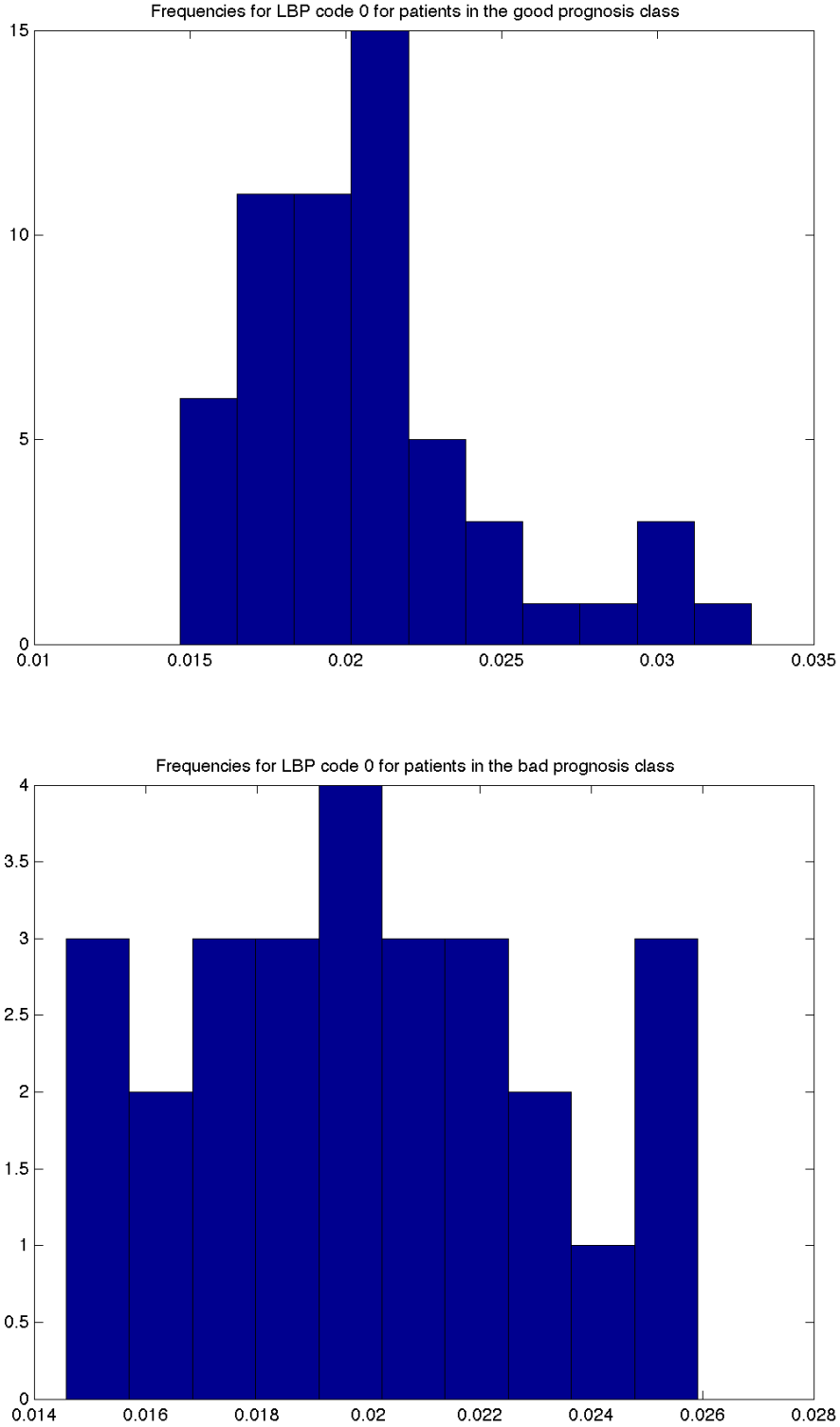

Figure D.7: Frequencies of LBP code 0 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom. Using all aviable patients. The bad prognosis class has the same values as in previous figure since there only are 27 patients in the bad prognosis class.

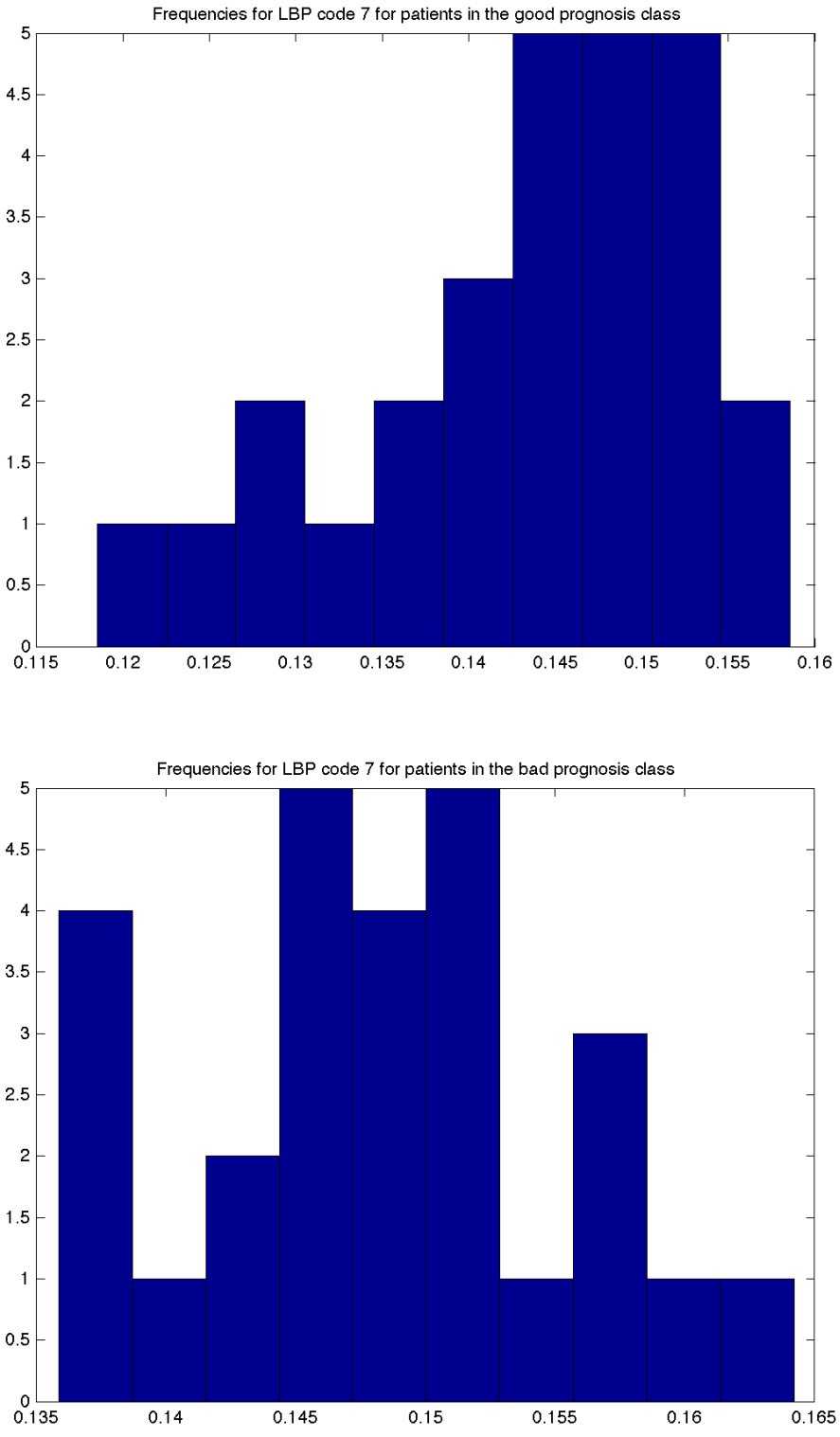

Figure D.8: Frequencies of LBP code 7 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom. Using 27 patients from each of the classes.

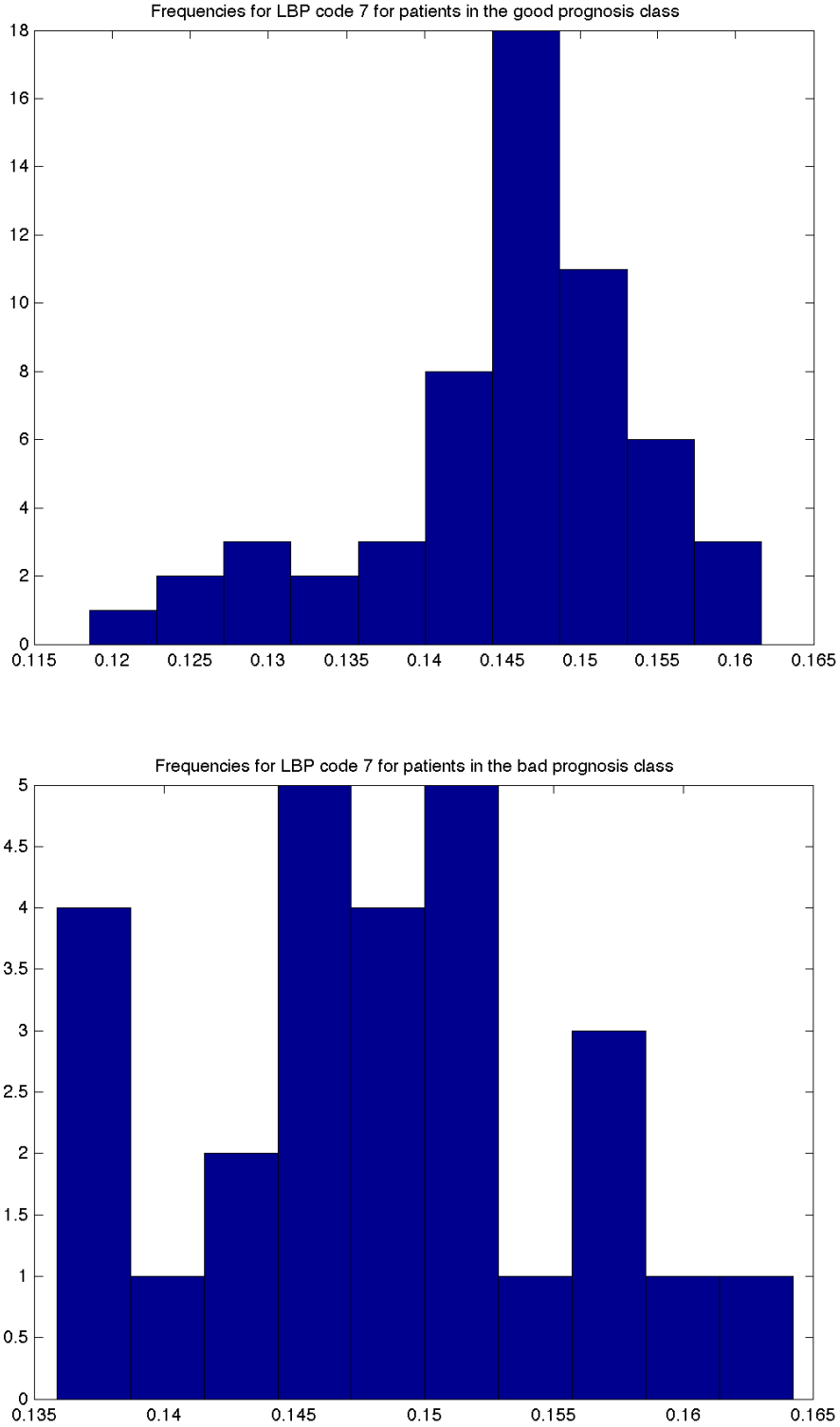

Figure D.9: Frequencies of LBP code 7 for patients within the two classes, good prognosis class at the top, bad prognosis class at the bottom. Using all aviable patients. The bad prognosis class has the same values as in previous figure since there only are 27 patients in the bad prognosis class.

Bibliography

[1] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel, and Distributed*. Course Technology, 2005.

[2] T. M. Cover. The best two indepentene measurements are not the two best. *IEEE Transactions on Systems, Man and Cybernetics*, (4):116–117, 1974.

[3] P. A. Devijver and J. Kittler. *Pattern Recognition - A Statistical Approach*. Prentice-Hall, 1982.

[4] D. W. Hedley. DNA analysis from paraffin-embedded blocks. *Methods Cell Biol.*, 41:231–240, 1994.

[5] H.Schulerud and F. Albregtsen. Many are called, but few are chosen. feature selection and error estimation in high dimensional spaces. *Computer Methods and Programs in Biomedicine*, (73):91–99, 2004.

[6] A. Jain and D. Zongker. Feature selection: Evaluation, application, and small sample performance. *IEEE Trans. on Pattern Analysis and Machine Intell.*, (19):153–158, 1997.

[7] G. B. Kristensen, W. Kildal, V. M. Abeler, J. Kaern, I. Vergote, C. G. Tropé, and H. E. Danielsen. Large-scale genomic instability predicts long-term outcome for women with invasive stage I ovarian cancer. *Annals of Oncology*, 14:1494–1500, 2003.

[8] T. Marill and C. M. Green. On the effectiveness of receptors in recognition systems. *IEEE Trans. Inform. Theory*, (9):11–17, 1963.

[9] B. Nielsen, F. Albregtsen, and H. E. Danielsen. Local binary pattern matrix - a new approach to nuclear texture analysis. *Analytical Cellular Pathology*, (25), 2003.

[10] B. Nielsen, F. Albregtsen, and H. E. Danielsen. Low dimensional adaptive texture feature vectors from class distance and class difference matrices. *IEEE Transactions on Medical Imaging*, 23(1):73–84, January 2004.

[11] B. Nielsen, F. Albregtsen, and H. E. Danielsen. Statistical nuclear texture analysis in cancer research: A review of methods and applications. *Critical Reviews in Oncogenesis*, 14:89–164, 2008.

[12] P. Pudil, J. Novovicova, and J. Kittler. Floating search methods in feature selection. *Pattern Recognition Letters*, (15):1119–1125, 1994.

[13] T. Randen and J. H. Husøy. Filtering for texture classification: A comparative study. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(4), April 1999.

[14] P. Somol and P. Pudil. Oscillating search algorithms for feature selection. *Proceedings. 15th International Conference on Pattern Recognition*, 2:406–409, 2000.

[15] P. Somol, P. Pudil, J. Novovičová, and P. Paclík. Adaptive floating search methods in feature selectionq. *Pattern Recognition Letters*, 20:1157–1163, 1999.

[16] H. J. Tanke and E. M. van Ingen. A reliable Feulgen-acriflavine-SO_2 staining procedure for quantitative DNA measurements. *J. Histochem. Cytochem. 28*, pages 1007–1013, 1980.

[17] A. Wayne Whitney. A direct method of nonparametric measurement selection. *IEEE Transactions on Computers*, pages 1100–1103, September 1971.

[18] T. Y. Young and K. FU. *Handbook of Pattern Recognition and Image Processing.* Academic Press, 1986.

[19] D. Zongker and A. Jain. Algorithms for feature selection. an evaluation, in: Proceedings. *IEEE Transactions on Systems, Man and Cybernetics*, (4):116–117, 1974.